ILOG Concert Technology 1.3

User's Manual

December 2002

Table of Contents

Welcome to ILOG Concert Technology

This user's manual explains how to get the most from ILOG Concert Technology, the means for modeling, modifying, and solving optimization problems. Concert Technology enables you to formulate optimization problems independently of the algorithms that you may eventually choose to solve the problem. It provides an extendable modeling layer adapted to a variety of algorithms ready to use off the shelf. This modeling layer enables you to change your model in a variety of ways, without rewriting your application.

What Is Concert Technology?

Concert Technology offers a C++ library of classes and functions that enable you to define models for optimization problems and to apply algorithms to those models. Concert Technology supports algorithms for both constraint programming and math programming (including linear programming, mixed integer programming, quadratic programming, and network programming) solutions. It also allows you to extend the model and solution classes yourself.

This library is not a new programming language: it lets you use data structures and control structures provided by C++. Thus, the Concert Technology part of an application can be completely integrated with the rest of that application (for example, the graphic interface, connections to databases, etc.) because it can share the same objects.

Furthermore, you can use the same objects to model your problem whether you choose a math programming or constraint programming approach. In fact, Concert Technology enables you to combine these technologies simultaneously.

Innovative Modeling in Concert Technology

The editing facilities in Concert Technology enable you to modify your model in a variety of ways. In formal terms, since the model in Concert Technology is cleanly separated from the search algorithms in other ILOG products, such as ILOG CPLEX and ILOG Solver, you can make *non monotonic changes* in your model without undermining the completeness of your search algorithms. In practical terms, you can make the following changes in a model:

- Extend the domain of a variable.
- Increase or decrease the bounds on a variable.
- Add variables to an initial Special Ordered Set (SOS).
- Add constraints to a logical OR (also known as a disjunctive constraint or an instance of `IloOr`).
- Change the coefficients of a linear expression.
- Add rows or columns to a linear model.
- Relax integrality constraints on variables.

What You Need to Know

This manual assumes that you are familiar with the operating system where you are using Concert Technology. Since Concert Technology is written for C++ developers, this manual assumes that you can write C++ code and that you have a working knowledge of your C++ development environment.

Notation

Throughout this manual, the following typographic conventions apply:

- Samples of code are written in this `typeface`.
- Important ideas are emphasized like *this*.

Naming Conventions

The names of types, classes, and functions defined in the Concert Technology library begin with `Ilo`.

The names of classes are written as concatenated, capitalized words. For example:

```
IloNumVar
```

A lower case letter begins the first word in names of arguments, instances, and member functions. Other words in such a name begin with a capital letter. For example,

`aVar`

`IloNumVar::getType`

There are no public data members in Concert Technology.

Accessors usually begin with the keyword `get` followed by the name of the data member. Accessors for Boolean members begin with `is` followed by the name of the data member. Like other member functions, the first word in such a name begins with a lower case letter, and any other words in the name begin with a capital letter.

Modifiers usually begin with the keyword `set` followed by the name of the data member.

Related Documents

The Concert Technology 1.3 library comes with:

- The reference manual, delivered with the standard distribution and accessible through conventional html browsers.
- This user's manual, delivered with the standard distribution. It shows you how to use Concert Technology by walking through examples.
- Source code for examples delivered in the standard distribution.

Other ILOG Products

If you have purchased licenses for additional ILOG products, such as ILOG CPLEX, ILOG Solver, ILOG Scheduler, ILOG Dispatcher, or ILOG Configurator, then the documentation in the standard distribution of those products includes the corresponding reference manuals and user's manuals for those licensed options.

Finding What You Need

Generally, classes and global functions with a name that begins `Ilo` support optimization *modeling*. In contrast, classes and global functions named `Ilc` generally support a constraint programming *search* for a solution. Names prefixed by `CPX` are C routines in the ILOG CPLEX library. There is a corresponding C++ member function in the class `IloCplex` for each of those C routines.

The *ILOG Concert Technology Reference Manual* documents primarily modeling objects and functions. The algorithms (derived from `IloAlgorithm`) to use with these models are documented in their respective product manuals.

- `IloCplex` is documented in the *ILOG CPLEX Reference Manual*.
- `IloSolver` is documented in the *ILOG Solver Reference Manual*.

For More Information

ILOG offers technical support, users' mailing lists, and comprehensive websites for its products, including ILOG Concert Technology, ILOG CPLEX, and ILOG Solver.

Customer Support

For technical support of Concert Technology, you should contact your local distributor, or, if you are a direct ILOG customer, contact:

Region	Email	Telephone	Fax
France	concert-support@ilog.fr	0 800 09 27 91 (numéro vert)	+33 (0)1 49 08 35 10
Germany	concert-support@ilog.de	+49 6172 40 60 33	+49 6172 40 60 -10
Japan	concert-support@ilog.co.jp	+81 3 5211 5770	+81 3 5211 5771
North America	concert-support@ilog.com	1 877 ILOG TECH (toll free) 1 650 567 8080	+1 650 390 0946
Singapore	concert-support@ilog.com.sg	+65 6773 06 26	+65 6773 04 39
Spain	concert-support@ilog.es	+34 902 170 295	+34 91 372 9976
United Kingdom	concert-support@ilog.co.uk	+44 1344 661630	+44 1344 661601

We encourage you to use e-mail for faster, better service.

Users' Mailing List

The electronic mailing list `concert-list@ilog.fr` is available for you to share your development experience with other Concert Technology users. This list is not moderated, but subscription is subject to an on-going maintenance contract. To subscribe to `concert-list`, send e-mail without any subject to `concert-list-owner@ilog.fr`, with the following contents:

```
subscribe concert-list
your e-mail address if different from the From field
first name, last name
your location (company and country)
maintenance contract number
maintenance contract owner's last name
```

Web Sites

There are two kinds of web pages available to users of Concert Technology: web pages restricted to owners of a paid maintenance contract; web pages freely available to all.

Web Pages for a Paid Maintenance Contract

The customer support pages on our world wide web sites contain FAQ (Frequently Asked/Answered Questions) and the latest patches for some of our products. Changes are posted in

the product mailing list. Access to these pages is restricted to owners of an on-going maintenance contract. The maintenance contract number and the name of the person this contract is sent to in your company will be needed for access, as explained on the login page.

All three of these sites contain the same information, but access is localized, so we recommend that you connect to the site corresponding to your location, and select the Services page from the home page.

- Americas: `http://www.ilog.com`
- Asia & Pacific Nations: `http://www.ilog.com.sg`
- Europe, Africa, and Middle East: `http://www.ilog.fr`

Web Pages for General Information

In addition to those web pages for technical support of a paid maintenance contract, you will find other web pages containing additional information about Concert Technology, including technical papers that have also appeared at industrial and academic conferences, models developed by ILOG and its customers, news about progress in optimization. This freely available information is located at these localized web sites:

- `http://www.ilog.com/products/optimization/times/`
- `http://www.ilog.com.sg/products/optimization/times/`
- `http://www.ilog.fr/products/optimization/times/`

Part I

Meet ILOG Concert Technology

This part of the manual introduces ILOG Concert Technology, particularly its facilities for modeling. It offers a method for creating an application that uses Concert Technology, and it describes some of its features and classes.

C H A P T E R

1

Working with Concert Technology

This chapter explains how to begin working with Concert Technology. In it, you will learn:

- some of the advantages of Concert Technology;
- how to build your own model using Concert Technology classes;
- how to extract your model for a Concert Technology algorithm;
- how to solve for a solution;
- how to access a solution.

Why Concert Technology?

When you are solving an optimization problem, a first step is to devise a model for the problem. A model consists of a set of decision variables, a set of constraints that the decision variables must satisfy, and optionally, an objective function to minimize or maximize. The second step of solving an optimization problem is to apply an algorithm to the model. Concert Technology provides C++ classes for representing models as well as the base classes for implementing algorithms.

ILOG Concert Technology is a *library* of C++ classes and functions. As such, it offers all the software engineering advantages of C++ and object-oriented programming. The library is extendable and readily integrated with other software applications.

ILOG offers two fundamentally different technologies for solving optimization problems: mathematical programming and constraint programming. Concert Technology provides a way for C++ programmers to use a consistent representation for applying both technologies. This consistency also makes it straightforward to develop algorithms where the linear programming and constraint programming technologies from ILOG share the work in order to find a solution as quickly as possible.

One Model, Many Algorithms

Often, after developing a model for a problem, the developer needs to experiment in order to find the best algorithm for solving a particular model. Because Concert Technology maintains separate data structures for models and algorithms, it is easy to experiment in that way. In addition, Concert Technology allows a developer to extract different parts of a model and apply different algorithmic technologies and search strategies to those different parts.

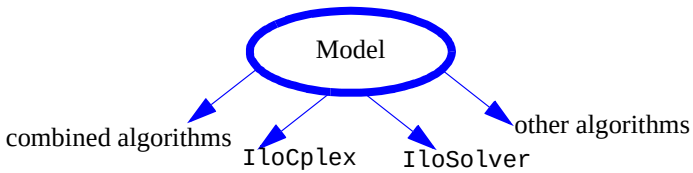

Figure 1.1 *One model extracted for many algorithms*

ILOG Solver, for example, supports specialized algorithms in conjunction with Concert Technology, for linear and non linear parts of a model. Similarly, Concert Technology offers an interface to ILOG CPLEX algorithms such as primal simplex, dual simplex, barrier, mixed integer, network, and quadratic.

Modifiable Models

After solving a given problem, a developer often wants to solve a related problem that represents a change from an existing model. Concert Technology has a set of classes and methods that allow developers to carry out these problem modifications. After a model is modified, the algorithms are notified of the changes and can take advantage of the previous solution in order to find a suitable solution to the modified problem.

In fact, Concert Technology allows monotonic changes in a model and thus supports certain complete algorithms that are sure to converge toward a solution. At the same time, it allows *non monotonic* change as well, thus supporting local optimizations and other incomplete but practical approaches to otherwise intractable problems. That is, the modeling facilities in Concert Technology allow users full editing capabilities of their models.

In this manual, the example of ILOG Scheduler in *Pareto-Optimization: Using ILOG Scheduler* is a clear example of modifying a model effectively in this way.

Models and Submodels

With Concert Technology, you may create more than one model in a given environment. (We'll have more to say later about Concert Technology environments in this chapter in *Creating an Environment* and in a later chapter in *The Environment: IloEnv*.) In fact, users can create *submodels*. That is, a user can add one model to another model within the same environment. This facility makes problem decomposition straightforward, and it also aids preprocessing and presolving as ways of mastering otherwise intractable problems. For example, the user may define one model to represent the restricted master problem (RMP) and another model to represent the subproblem.

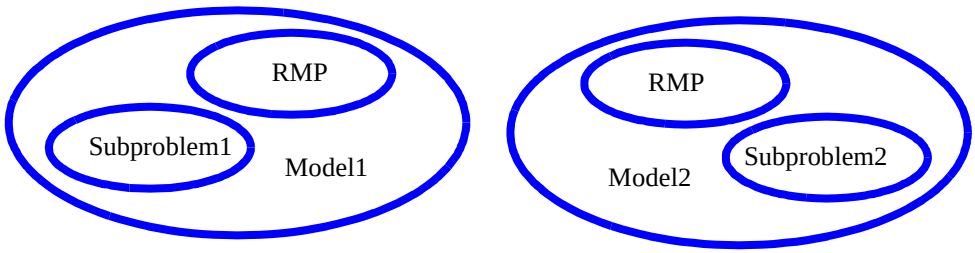

Figure 1.2 *Reusable subproblems appear in different models*

In this manual, the example of column generation, in *Cutting Stock: Column Generation* shows an effective use of an initial submodel incremented as needed.

File Formats and Model Sharing

Because the data in models are cleanly separated from its algorithms, Concert Technology can use various file formats for representing optimization problems. Conventional linear programming formats (such as MPS, CPLEX SAV, and CPLEX LP) are supported, allowing problems to be represented as C++ code as well as text in files. This diversity assists developers in debugging their models. It also provides a means of sharing models between developers and ILOG support personnel.

Lightweight

ILOG has devoted many development years to making Concert Technology models lightweight and supple with respect to memory consumption. We've gathered best practices from diverse programming communities, such as operations research, math programming, and constraint programming, to master the software engineering issues underlying an

intuitive and natural approach to modeling. Users of Concert Technology will be able to design a model of their complete problem with less concern about managing memory allocation themselves or encountering resource exhaustion before they achieve a satisfactory solution.

Solutions

In their various approaches to mastery of intractable, real-world problems, markedly different ways of defining solutions have evolved in the math programming and constraint programming communities. Concert Technology offers facilities for finding, expressing, and accessing solutions that support both communities. The class `IloCplex`, the base class for math programming approaches (LP, MIP, and so forth), for example, offers member functions for analyzing and measuring the feasibility of a solution. The class `IloSolver`, the base class for constraint programming approaches, offers member functions to manage provably safe solutions, complete solutions, and just good enough solutions.

The solution facilities in Concert Technology are also extendable. In fact, other ILOG products extend these facilities to support solutions adapted to particular problem domains. ILOG Scheduler, for example, offers a solution object adapted to constraint-based scheduling and resource allocation. Likewise, ILOG Dispatcher offers objects adapted to the solution of problems in vehicle routing and technician dispatching. ILOG Configurator offers solution objects that can be declared and used before their details are fully specified by the resolution of a configuration problem.

Concert Technology also supports partial resolution and iterative resolution of problems. In this manual, the example of ILOG Dispatcher in *Dispatching Technicians: Using ILOG Dispatcher* straightforwardly demonstrates an iterative resolution of a problem. For more examples of those approaches, see the user's manuals of the various ILOG optimization products.

A Concert Technology Application

A typical application using Concert Technology consists of these steps:

- Creating an Environment
- Building a Model
- Extracting a Model for an Algorithm
- Solving the Problem
- Accessing Results
- Ending the Program

Later chapters of this manual illustrate those steps in detail in actual applications. The following sections sketch the general idea of each of those steps.

Creating an Environment

The first step in a Concert Technology application is to create the *environment*, an instance of the class `IloEnv`, to manage memory in the Concert Technology part of your application. Typically, we do so early in the `main` part of an application, like this:

```
IloEnv env;
```

Every Concert Technology model and every algorithm must belong to an environment. In C++ terms, when you construct a model (an instance of the class `IloModel`) or an algorithm (an instance of one of the subclasses of `IloAlgorithm`, such as `IloSolver` or `IloCplex`, for example), then you must pass one instance of `IloEnv` as a parameter of that constructor. Concert Technology objects that are constructed with an instance of `IloEnv` as a parameter will be managed automatically with respect to memory allocation and de-allocation in your application.

Environment and Communication Streams

An instance of `IloEnv` in your application initializes its default streams (`ostreams`) for general information, for warnings, and for error messages. Each environment maintains its own communication streams.

Environment and Memory Management

When your application deletes an instance of `IloEnv`, Concert Technology will automatically delete all models and algorithms depending on that environment as well. As you will see repeatedly in the examples in this manual, we recommend that you use the member function `IloEnv::end`, documented in the *ILOG Concert Technology Reference Manual*, to delete an environment.

Building a Model

Within an environment, we then conventionally create a *model*, an instance of the class `IloModel`, like this:

```
IloModel model(env);
```

Initially, at the moment that you declare it, a model does not yet contain the objects (such as constraints, constrained variables, objectives, and possibly other modeling objects) that represent your problem. You will populate that model by adding objects to it, as we explain later in *Adding Objects to a Model*.

Later in your application, when you create an algorithm, that Concert Technology algorithm extracts information from your model and uses the information in an appropriate form to solve the problem. Various Concert Technology algorithms can extract diverse relevant information from the same model.

Extractable Objects in a Model

Objects—such as constraints, constrained variables, objectives, columns, ranges, numeric variables, and so forth—that play a role in your model are represented in Concert Technology by instances of the class `IloExtractable` or one of its subclasses. A model itself is such an object, making it possible for you to organize objects into subproblems and to combine subproblems into more complicated models.

Adding Objects to a Model

Given a model (an instance of `IloModel`) in an environment (an instance of `IloEnv`), you can add objects to your model in several different ways:

- You can use the template `IloAdd`, like this:

```
IloObjective obj = IloAdd(model, IloMinimize(env));
```

- You can use the member function `IloModel::add`, like this:

```
model.add(IloSum(x) == demand);
```

Seeing those alternative ways of adding objects to a model, you might well ask when to use one rather than the other. In that context, it is worth noting that `IloAdd` preserves the original type of its second argument when it returns. This feature of the template may be useful, for example, in cases like this:

```
IloRange rng = IloAdd(model, 3 * x + y == 17);
```

In other words, in programming situations where you need to access or to preserve the type of an object, then you should use `IloAdd`.

Concert Technology is capable of extracting appropriate information from such objects in your model for use by your application in Concert Technology algorithms. We refer to this process as *extraction*, and we call such objects *extractable*.

Extracting a Model for an Algorithm

All the extractable objects (that is, instances of `IloExtractable` or one of its subclasses) that have been added to a model (an instance of `IloModel`) and that have not been removed from it and that are relevant to the given algorithm will be extracted when a Concert Technology algorithm extracts information from a model.

```
cplex.extract(model);
```

In case you need to access the extractable objects in a model, an instance of the embedded class `IloModel::Iterator` accesses those objects. For example, the following lines create the model `model` in the environment `env`, add constraints to the model, and iterate over the extractable objects added to `model`.

```
for (IloModel::Iterator it(model); it.ok(); ++it) {
    if ((*it).getName())
        env.out() << (*it).getName() << endl;
}
```

Solving the Problem

To solve a problem with Concert Technology, you need to indicate which algorithm you want to use. You do that by creating an instance of one of the subclasses of `IloAlgorithm`, such as `IloSolver` or `IloCplex`, in your environment. Then you call the `extract` member function of the algorithm. Then you call the `solve` member function for that algorithm.

For example, to use the ILOG CPLEX algorithm (an instance of `IloCplex`), you do this:

```
IloCplex cplex(env);
cplex.extract(model);
cplex.solve();
```

Accessing Results

As you will see in the examples later in this manual, Concert Technology supports a variety of ways of accessing results.

Displaying a Solution in *Facility Planning: Using Concert Technology* shows how to use the output operator << with the member functions `solver::out` and `solver::getValue` (whether the `solver` is an instance of `IloCplex` or `IloSolver`) to display the results of a search.

Complete Program in *Cutting Stock: Column Generation* shows how to use your own, user-defined reporting functions to indicate progress toward a solution.

Displaying a Solution in *Transport: Piecewise Linear Optimization* shows how to use the member functions `IloEnv::out` and `IloCplex::getValue` to display the results of a search.

Displaying a Solution in *Car Sequencing: Using ILOG Solver* shows how to use the member function `IloSolver::printInformation` to display technical details about the search, such details as the number of choice points, the number of constraints, and so forth.

Displaying a Solution in *Dispatching Technicians: Using ILOG Dispatcher* shows the definition of a user-written function combining member functions from classes of both ILOG Solver and Dispatcher to access information about a solution.

Displaying a Solution in *Filling Tankers: Using ILOG Configurator* shows how to iterate over a *set* of solutions.

Displaying a Solution in *Pareto-Optimization: Using ILOG Scheduler* shows how to use a user-written function to iterate over the elements of a *schedule* in a solution.

Ending the Program

When your application no longer needs the model(s) and algorithm(s) that you have constructed with Concert Technology, then you end that part of the program by calling the member function `IloEnv::end`. That member function de-allocates the Concert Technology objects, such as constrained variables, ranges, objective, models, algorithms, search objects, and so forth, allocated in the invoking environment.

```
env.end();
```

After that call of `env.end`, any object created with that instance of `IloEnv` as a parameter will be de-allocated; that is, its memory will be freed, and the object will no longer be available for use in your application.

Programming Hint: Try and Catch

In the examples in this manual, you will see that the applications include one `try{}` and `catch{}` to intercept exceptions thrown by Concert Technology. We urge you to follow this programming practice.

Programming Hint: Managing Memory Yourself

As we mentioned, a call of the member function `IloEnv::end` de-allocates the Concert Technology objects, such as constrained variables, ranges, objective, models, algorithms, search objects, and so forth, allocated in the invoking environment. If you want to implement your own, more finely grained memory management, for example, in a large-scale application with considerable iteration, Concert Technology also provides the means for that through `end` member functions in its classes. For example, you can use the member function `IloExpr::end`, documented in the *Concert Technology Reference Manual*, to delete temporary expressions when they are no longer in use in your application. Similar facilities are also available for temporary arrays. This optional memory management is useful in situations where you judge that reclaiming temporary memory may be more

advantageous than relying on the automatic memory management offered by `IloEnv::end`. There is an example of this reclamation of memory allocated to a temporary expression in this manual in *Programming Practices*. There is a similar demonstration in the example of the magic squares (`magicsq.cpp`) explained in the *ILOG Solver User's Manual*.

C H A P T E R

2

Facility Planning: Using Concert Technology

This chapter presents a facility planning problem in ILOG Concert Technology. In it, you will learn:

- how to read data from a file into arrays in your model;
- how to represent problem constraints in your model;
- how to add an objective function to your model;
- how to use the same model for various algorithms;
- how to extract a model for an algorithm (an instance of `IloSolver` or `IloCplex`, for example) and use it to solve a problem;
- how to display a solution.

Describing the Problem

In this facility planning problem, we assume that there are a number of locations where we could open warehouses or other facilities to serve clients. There is an initial fixed cost associated with opening each facility. There is also a known cost associated with serving a given client from one facility rather than from another facility. Each facility has a limited

capacity; that is, it can serve only so many clients. We also stipulate that each client will be served by only one facility. Within these assumptions, we want to find the least costly choice of locations where we should open a facility in order to meet our clients' demands; we also want to know which facility should serve each client.

Creating an Environment

As a first step in developing this application, we create a Concert Technology environment, an instance of `IloEnv`, to manage memory allocations and deletion of Concert Technology objects.

```
IloEnv env;
```

In that environment, we create several arrays of integer variables. These Concert Technology arrays of constants or variables (instances of `IloIntVarArray`) are extensible. They can also be organized into matrices, that is, arrays of arrays. We use symbolic values, such as `nbClients` to indicate the number of clients or `nbLocations` to indicate the number of locations, in order to take advantage of that extensibility.

- The array `capacity` contains one value for each facility. A value of that array indicates the capacity of a facility.
- The two-dimensional array supply contains one binary variable for each couple (client, location). The value of a variable from `supply` indicates whether a client is served by a location.
- The array `open` contains one element for each location. Its variables will indicate whether or not we should open a facility at the corresponding location.

```
IloIntVarArray open(env, nbLocations, 0, 1);
IntVarMatrix supply(env, nbClients);
for(i = 0; i < nbClients; i++)
  supply[i] = IloIntVarArray(env, nbLocations, 0, 1);
```

Representing the Data

Concert Technology accepts problem data from many sources. Your application, for example, might generate problem data itself or perhaps retrieve it from a database or accept input from another application. In any case, this example shows you how to read problem data from a file into arrays in ways that you can adapt to your own application.

Reading Data from a File into Arrays

Since each of the facilities has a known fixed capacity, we will represent that data as `capacity`, an array of numeric values. Each element of the array corresponds to the capacity of one facility. We will use another array of numeric values, `fixedCost`, to represent the corresponding fixed cost of opening a facility.

```
IloNumArray capacity(env), fixedCost(env);
```

Using Multidimensional Arrays

You can also create multidimensional arrays in Concert Technology. In a problem such as this one, where the cost varies for using one facility rather than another to serve a given client, we can readily represent the data as a two-dimensional array or matrix, `cost`. In fact, Concert Technology provides a template type definition to facilitate your creation of multidimensional arrays.

```
typedef IloArray<IloIntVarArray> IntVarMatrix;
```

Using Extensible Arrays

That declaration creates an array of length zero. Interestingly, Concert Technology offers *extensible* arrays. That is, you may add more elements to them even after their initial declaration, whether they are arrays of numeric or integer values (that is, instances of `IloNumArray` or `IloIntArray`) or arrays of decision variables (such as instances of `IloIntVarArray`, `IloBoolVarArray`, or `IloNumVarArray`). You can also shorten these extensible arrays by removing elements from them.

The template `IloArray` defines the base class of the predefined array classes, such as `IloNumArray` and `IloNumVarArray`, documented in the *Concert Technology Reference Manual*. That template also offers you a means of defining your own extensible arrays adapted to features of your own Concert Technology application.

One way of extending an array is to read data from a file by means of the input operator, like this:

```
file >> capacity >> fixedCost >> cost;
    nbLocations = capacity.getSize();
    nbClients   = cost.getSize();
```

The template for extensible arrays also defines member functions accessing useful information about arrays. For example, after we read data from a file into extensible arrays in those lines, we can call the member function `getSize` to tell us the length of the arrays. There is also a `remove` member function for shortening an array by removing elements.

Determining Length of Extensible Arrays

In this problem, we make the number of sites available for opening facilities a symbolic value, `nbLocations`, that we can determine from the data we read from our input file. Likewise, we also make the number of clients a symbolic value, `nbClients`, that we determine from reading the input file as well.

We put together these ideas—extensible arrays, multidimensional arrays, user-defined arrays—in these lines of code for reading data from an input file.

```
const char* filename  = "../../../examples/data/facility.dat";
    if (argc > 1)
  filename = argv[1];
ifstream file(filename);
if (!file) {
  cerr << "usage:   " << argv[0] << " <file>" << endl;
  throw FileError();
}

file >> capacity >> fixedCost >> cost;
```

Checking Input

As a careful programming practice, we use `getSize` again, this time to check whether the data file we read contains a fixed start-up cost for every location as well as a capacity for each. We also check whether for each client, there is a known cost for serving that client from each of the possible locations. In each of those checks, we use the operator == to compare the size of arrays. In short, are the corresponding arrays of the same length? Is at least one dimension of the matrix equal to the number of locations?

```
file >> capacity >> fixedCost >> cost;
    nbLocations = capacity.getSize();
    nbClients   = cost.getSize();

IloBool consistentData = (fixedCost.getSize() == nbLocations);
for(i = 0; consistentData && (i < nbClients); i++)
  consistentData = (cost[i].getSize() == nbLocations);
    if (!consistentData) {
  cerr << "ERROR: data file '"
               << filename << "' contains inconsistent data" << endl;
  ;
}
```

Developing a Model

Now we create a model within the same environment and add the problem constraints to it.

- Only one facility will supply a given client.
- The sum of clients that a facility supplies must not exceed the capacity of that facility.
- A facility can serve a given client only if that facility is open.

```
IloModel model(env);
for(i = 0; i < nbClients; i++)
  model.add(IloSum(supply[i]) == 1);
for(j = 0; j < nbLocations; j++){
  IloExpr v(env);
  for(i = 0; i < nbClients; i++)
    v += supply[i][j];
  model.add(v <= capacity[j] * open[j]);
  v.end();
    }
```


Adding an Objective to the Model

To compute an objective function in this example, we need to consider:

- which facilities to open; this idea is represented by binary decision variables in the array `open`;
- the fixed cost entailed by each open facility; this value appears in the array `fixedCost`;
- the cost for serving `client i` from `facility j`; this idea appears in the array `supplyCost`; its elements are filled in like this:
  `supplyCost[i] == cost[i](supplier[i])`
- the accumulated costs for supplying all clients from open facilities.

To express the objective function, we use the predefined Concert Technology functions `IloSum` to compute the sum of elements of an array and `IloScalProd` to compute the scalar product of the elements of two arrays. Both those functions can operate on integer or numeric *variables* as well as on ordinary integer or numeric values.

We then indicate the sense (a minimization in this case) of the objective function and add it to our model, like this:

```
IloExpr obj = IloScalProd(fixedCost, open);
for(i = 0; i < nbClients; i++){
  obj += IloScalProd(cost[i], supply[i]);
    }
model.add(IloMinimize(env, obj));
```

Solving the Problem

With the constraints we have added to the model, we can solve the problem in three lines of Concert Technology code. We simply create an algorithm in the environment, extract the model for that algorithm, and solve.

Using IloCplex

For example, if we want to solve our problem with an instance of `IloCplex`, documented in the *ILOG CPLEX Reference Manual*, we do so like this:

```
IloCplex cplex(env);
cplex.extract(model);
cplex.solve();
```

An instance of `IloCplex` is clearly a good choice when the model of the problem is a linear program (LP); that is, when the model contains only purely linear constraints, when there are no strict inequalities, and when there are only continuous variables. The *ILOG CPLEX User's Manual* contains a host of programming examples demonstrating how to use `IloCplex` to best advantage in those cases.

When the model represents a purely integer or mixed integer problem, the choice is more subtle since `IloCplex` offers good MIP facilities, too. In those cases, when the linear relaxation is tight, (for example, in production planning problems), `IloCplex` is a good choice. Again, the *ILOG CPLEX User's Manual* contains examples showing how to design your model and tune your search using `IloCplex` in those situations.

When the linear relaxation of a MIP model is *not* tight, (for example, in scheduling problems), then constraint programming may offer a better approach.

Using IloSolver

This same model of the problem can also be solved by an instance of `IloSolver`, documented in the *ILOG Solver Reference Manual*. We do so like this:

```
IloSolver solver(env);
solver.extract(model);
solver.solve();
```

As we noted, an instance of `IloSolver` is a good choice when the model of the problem is *not* clearly a linear program; that is, when the model includes non linear constraints, strict inequalities, or discontinuous variables. It is also a good choice in a purely integer model or in a mixed integer model when the linear relaxation is *not* tight. Furthermore, an instance of `IloSolver` is a reasonable choice when the model contains metaconstraints, global constraints, and other kinds of logical constraints. For example, constraints that entail

abstraction, distribution, or sequences of variables suggest solution by an instance of `IloSolver`. The *ILOG Solver User's Manual* includes many examples of such problems and models.

Using Linear Programming with Constraint Programming: a Cooperative Method

Many problems include both linear and non linear components in their model. Concert Technology offers you the means to combine both linear programming and constraint programming techniques in tackling the same problem. To pursue this hybrid approach, you use an instance of `IloSolver` followed by an instance of `IloLinConstraint`. (`IloLinConstraint` is documented in the *ILOG Hybrid Optimizers User's Guide & Reference*.) You then extract and solve your model in the usual way, like this:

```
IloSolver solver(env);
IloLinConstraint lc(solver);
solver.extract(model);
solver.solve();
```

In short, you need exactly one more line of code to tap this hybrid approach in your model.

Displaying a Solution

To display the solution, we use the output operator and the member function `getValue`. For example, if we have used an instance of `IloSolver` to solve our problem, then we display those results like this:

```
solver.out() << "Optimal value: " << solver.getValue(obj) << endl;
 for(j = 0; j < nbLocations; j++){
   if (solver.getValue(open[j])){
     solver.out() << "Facility " << j << " is open, it serves clients ";
     for(i = 0; i < nbClients; i++){
       if (solver.getValue(supply[i][j]) == 1)
         solver.out() << i << " ";
     }
     solver.out() << endl;
   }
 }
     solver.printInformation();
}
```

If we have used an instance of `IloCplex` for our solution, we display the solution in a similar way, this time indicating a tolerance for values, like this:

```
    IloNum tolerance = cplex.getParam(IloCplex::EpInt);
    cplex.out() << "Optimal value: " << cplex.getObjValue() << endl;
    for(j = 0; j < nbLocations; j++){
      if (cplex.getValue(open[j]) >= 1 - tolerance){
        cplex.out() << "Facility " << j << " is open, it serves clients ";
            for(i = 0; i < nbClients; i++){
          if (cplex.getValue(supply[i][j]) >= 1 - tolerance)
            cplex.out() << i << " ";
                }
        cplex.out() << endl;
          }
    }
  }
```

If we use an instance of `IloLinConstraint` with `IloSolver`, we again indicate a tolerance for values in our display of a solution. In other respects, the display is much the same.

Ending an Application

To indicate to Concert Technology that our application has completed its use of the environment, we call the member function `IloEnv::end`. This member function cleans up the environment, de-allocating memory used in the model and the search for a solution.

```
env.end();
```

Complete Program

You can see the complete program here or on line in the standard distribution of the products ILOG CPLEX, ILOG Solver, or ILOG Hybrid Optimizer in the files `/examples/src/facility_cplex.cpp`, `/examples/src/facility_solver.cpp`, or `/examples/src/facility_hybrid.cpp`. To run the example, you need licenses for ILOG Solver and ILOG CPLEX.

```
// ------------------------------------------------------------ -*- C++ -*-
// File: facility_solver.cpp
// --------------------------------------------------------------------------
//  Copyright (C) 1999-2000 by ILOG.
//  All Rights Reserved.
//
//  N O T I C E
//
//  THIS MATERIAL IS CONSIDERED A TRADE SECRET BY ILOG.
```

```
//  UNAUTHORIZED ACCESS, USE, REPRODUCTION OR DISTRIBUTION IS PROHIBITED.
// ---------------------------------------------------------------------------
//

#include <ilsolver/ilosolver.h>

ILOSTLBEGIN

class FileError : public IloException {
public:
  FileError() : IloException("Cannot open data file") {}
};

typedef IloArray<IloIntVarArray> IntVarMatrix;

int main(int argc, char **argv){
  IloEnv env;
  try{
    IloInt i, j;

    IloNumArray capacity(env), fixedCost(env);
    IloNumArray2 cost(env);
    IloInt      nbLocations;
    IloInt      nbClients;

    const char* filename  = "../../../examples/data/facility.dat";
        if (argc > 1)
      filename = argv[1];
    ifstream file(filename);
    if (!file) {
      cerr << "usage:   " << argv[0] << " <file>" << endl;
      throw FileError();
    }

    file >> capacity >> fixedCost >> cost;
        nbLocations = capacity.getSize();
        nbClients   = cost.getSize();

    IloBool consistentData = (fixedCost.getSize() == nbLocations);
    for(i = 0; consistentData && (i < nbClients); i++)
      consistentData = (cost[i].getSize() == nbLocations);
        if (!consistentData) {
          cerr << "ERROR: data file '"
               << filename << "' contains inconsistent data" << endl;
      ;
    }

    IloIntVarArray open(env, nbLocations, 0, 1);
    IntVarMatrix supply(env, nbClients);
    for(i = 0; i < nbClients; i++)
      supply[i] = IloIntVarArray(env, nbLocations, 0, 1);
```

```
    IloModel model(env);
    for(i = 0; i < nbClients; i++)
      model.add(IloSum(supply[i]) == 1);
    for(j = 0; j < nbLocations; j++){
      IloExpr v(env);
      for(i = 0; i < nbClients; i++)
        v += supply[i][j];
      model.add(v <= capacity[j] * open[j]);
      v.end();
        }

    IloExpr obj = IloScalProd(fixedCost, open);
    for(i = 0; i < nbClients; i++){
      obj += IloScalProd(cost[i], supply[i]);
        }
    model.add(IloMinimize(env, obj));

    IloSolver solver(env);
    solver.extract(model);
    solver.solve();

    solver.out() << "Optimal value: " << solver.getValue(obj) << endl;
    for(j = 0; j < nbLocations; j++){
      if (solver.getValue(open[j])){
        solver.out() << "Facility " << j << " is open, it serves clients ";
        for(i = 0; i < nbClients; i++){
          if (solver.getValue(supply[i][j]) == 1)
            solver.out() << i << " ";
        }
        solver.out() << endl;
      }
    }
        solver.printInformation();
  }
  catch(IloException& e){
    cerr  << " ERROR: " << e.getMessage() << endl;
  }
  env.end();
  return 0;
}
```

C H A P T E R

3

Getting to Know the Classes in Concert Technology

Following the introduction to a working method in *Working with Concert Technology*, and an illustration of that method in *Facility Planning: Using Concert Technology*, this chapter offers you a better acquaintance with the C++ classes available in ILOG Concert Technology. In it, you will learn:

- more about the basic classes available in the Concert Technology library;
- a few programming hints appropriate for a Concert Technology application.

This chapter does not take the place of the *ILOG Concert Technology Reference Manual*, but we hope it makes that manual easier to use by offering you a quick orientation and more detail than the method sketched in *Working with Concert Technology*.

The Environment: IloEnv

Every user of ILOG Concert Technology will become familiar with the class `IloEnv`, documented in the *ILOG Concert Technology Reference Manual*. An instance of this class represents a Concert Technology environment. The environment manages memory and other bookkeeping tasks associated with a an optimization problem.

Among other tasks, it manages memory for data structures (such as Concert Technology arrays and sets), for model objects (such as instances of `IloModel` and other extractable objects), and for the Concert Technology algorithms (that is, instances of subclasses of `IloAlgorithm`, such as `IloCplex` and `IloSolver`).

Other constructors in Concert Technology require an environment as a parameter, so an instance of `IloEnv` is normally the first object that you create in a Concert Technology application.

Creating an environment is as simple as declaring a C++ variable, as we showed in *Creating an Environment*:

```
IloEnv env;
```

After such a statement in your application, you can use that environment, for example, in passing it as a parameter to other constructors or functions.

Handles

Though an environment requires a certain amount of memory, you will find that when you assign an environment directly or pass an environment by value to a function, it is as efficient as an assignment of a C++ pointer. The concept of *handles* underlies this efficiency. Most documented Concert Technology classes (whether they represent your problem data, your model, or your algorithm) are handle classes. Instances of handle classes are merely pointers, but they offer an application programming interface (API) to normal C++ classes, the underlying implementation classes of the handles. When you assign one handle to another, you are in effect assigning that pointer.

As we hinted, the pointer of a handle points to another object, known as the *implementation* object. Its class, the implementation class, is the C++ class that actually implements all the functionality apparent in the API. When you call a member function of a handle, you are effectively calling a member function of the implementation object that the handle points to.

In practical terms, the concept of handles and implementation objects means that you can use Concert Technology in your applications primarily by passing pointers, without concern for the usual C++ syntax of ampersands and asterisks.

IloEnv as a Handle

An environment, that is, an instance of `IloEnv`, is unique among Concert Technology objects in that its default constructor creates both the handle and implementation object that the handle points to.

In contrast, other default constructors in Concert Technology are empty constructors, also known as null constructors. They create empty handles, also known as zero-pointers or 0-handles; that is, handles that do not point to an implementation object, but rather point to

0 (zero). Like other C++ pointers, you must initialize an empty handle before you use it, for example, by assigning another handle to it.

Empty Handles

You can query whether a handle is empty or not by means of its member function `getImpl`. That member function returns a pointer to the implementation object of the invoking handle. It returns 0 (zero) if the handle is empty. The member function `getImpl` is the only member function that you can safely call on an empty handle.

If you assign one handle to another, then two (or more) handles point to the same implementation object. In designing Concert Technology, we made an engineering decision that it would be prohibitively costly and inefficient to track automatically when an implementation object can be safely deleted, and we recognized that in some cases, it would not be possible for the library to track such information in your application. Consequently, it is up to the user to delete implementation objects explicitly. You do so by means of the member function `end`.

IloEnv and Memory Management

As we noted in *Ending the Program*, you must call `IloEnv::end` to delete an environment when it is no longer in use in your application. Normally, the last operation of a Concert Technology application is this call to delete the environment:

```
env.end();
```

This call will also delete the modeling objects that have been created in this environment; in other words, those modeling objects created with this environment as a parameter. The call will also free the memory involved in algorithms (instances of the subclasses of `IloAlgorithm`, such as `IloCplex` or `IloSolver`) created with this environment as a parameter.

It is possible to create more than one environment in an application, though there is generally no need to do so. (The chief exception to this rule occurs when you are creating distinct models in different threads of a multi thread application.) Concert Technology objects created in different environments cannot be used together. Conventionally, in this manual, we assume that there is only one instance of `IloEnv` in an application, and we refer to it as “the environment.”

Arrays and Other Concert Technology Data Structures

Concert Technology does not depend on any particular representation of data in a problem. Instead, it offers various classes for managing data:

- expressions (instances of `IloExpr` or one of its subclasses),
- arrays (created by the template `IloArray`),
- columns (an instance of `IloNumColumn`),
- sets (such as an instance of `IloNumSet`);
- tuples and sets of tuples (such as instances of `IloAnyTupleSet` or `IloNumTupleSet`).

Each of the classes mentioned in that list of data structures is documented more fully in the *ILOG Concert Technology Reference Manual*. This section highlights the chief features of Concert Technology arrays to give you a feel for these data structures.

To accept numerical data as input, for example, Concert Technology offers the class `IloNumArray`. An instance of this class is an array adapted for numerical data. You create an empty numerical array in this way:

```
IloNumArray data(env);
```

In that declaration, the parameter `env` insures that the empty array is not an empty handle.

Arrays Are Somewhat Like Handles: Initializing

Concert Technology arrays resemble handles in this respect: you can call the member function `getImpl` to determine whether an array has been initialized already. This convention allows you to write functions that optionally accept 0 (zero) or an array as a parameter. For example,

```
f(IloNumArray array=0);
```

Also you must initialize arrays before you use them. Only assignment and the member function `getImpl` can be safely used on an empty array. In the examples in this manual, we assume that arrays have been initialized before use.

Arrays Have Bells and Whistles

If compiled in debug mode, Concert Technology arrays can perform bound checking when you access elements of the array. The concept *Assert and NDEBUG* documented in the *ILOG Concert Technology Reference Manual* explains how to turn on and off this feature.

A Concert Technology array knows its own size. You can use the member function `getSize` to learn the length of the invoking array. For example, the following lines set all elements of the array data to the value 0.0:

```
for (IloInt i=0; i < data.getSize(); ++i)
     data[i] = 0.0;
```

You will see other samples using `getSize` in later chapters of this manual, for example, in *Determining Length of Extensible Arrays*.

As you see in *Using Extensible Arrays*, arrays in Concert Technology are extensible. The size of a Concert Technology array is not fixed; you can add elements. For example, the following line adds five elements to the array `data`, and each of those elements has the value 1.0:

```
data.add(5, 1.0);
```

You can also remove elements. For example, the following line starts at the element indexed by 2 and removes three elements from the array `data`:

```
data.remove(2, 3);
```

When you remove elements in that way, the elements following the removal are moved forward in the array. After a removal like that, the remaining elements in the array are numbered from 0 (zero) through `getSize()-1`. In other words, removing elements from an array in this way does not create holes in the array, but changes its size.

Input and Output: Reading and Writing Arrays

The C++ input and output operators, `operator >>` and `operator <<`, are defined for Concert Technology arrays. For example, if the array data consists of the elements 1.0, 2.0, and 3.0, then we can write it to the standard output stream like this:

```
cout << data;
```

And we will see results like this:

```
 [1.0, 2.0, 3.0]
```

Conventionally, the output of an array begins with a leading square bracket `[`. Elements are separated by commas. A closing square bracket `]` marks the end of the array. You can rely on the same format to read an array from a stream. For example, this line will read from standard input into the array `data`:

```
cin >> data;
```

User-Defined Classes of Arrays

Concert Technology provides a template class that enables you to define your own classes of arrays with the same powers as predefined classes of Concert Technology arrays. The template class is `IloArray`.

If you want to define multidimensional arrays such as `IloArray<IloNumArray>`, you can also use the template class `IloArray`. In other chapters of this manual, you will see repeated examples of this use of the template class, for example, in creating two-dimensional arrays in *Using Multidimensional Arrays*.

In order for the input and output operators to function on your template-defined class of arrays, those operators must be defined on the type of the elements of the template-defined array. That is, if you use `IloArray` to define a class of arrays for your user-defined type `MyClass`, then if the input and output operators are defined for instances of `MyClass`, they will also be defined by the template class `IloArray` for instances of `IloArray<MyClass>`.

Extractable Objects: IloExtractable

Thus far, this chapter has concentrated on a description of the *data structures* available in Concert Technology, and it has merely hinted about the modeling facilities and algorithms. You will find more detailed explanations of the algorithms in the reference and user's manuals of the respective ILOG products. For example, the algorithm class `IloCplex` is documented in the *ILOG CPLEX Reference* and *User's Manuals*, and similarly `IloSolver` is documented in the *ILOG Solver Reference* and *User's Manuals*. Here we will delve a little further into the *modeling* facilities of Concert Technology, particularly its classes of extractable objects.

The term extractable object comes from the fact that an algorithm (that is, an instance of one of the subclasses of `IloAlgorithm`, such as `IloCplex` or `IloSolver`) extracts pertinent information from the modeling object in a form appropriate to that algorithm for use in solving the problem.

All extractable classes are derived from a common base class `IloExtractable`, documented in the *ILOG Concert Technology Reference Manual*. Concert Technology provides a wealth of extractable classes that allow you to define variables, ranges, columns,

various kinds of constraints, and objective functions. The most prominent extractable classes appear in Figure 3.1.

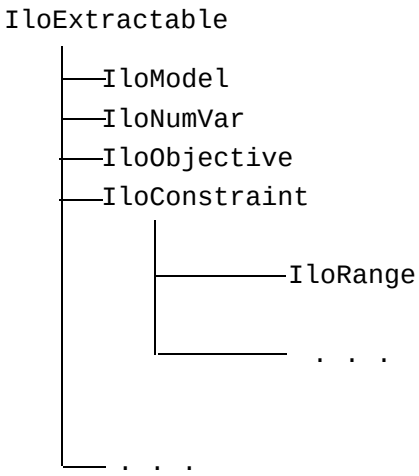

Figure 3.1 *Concert Technology classes derived from IloExtractable*

You can construct an extractable object only within an environment; in other words, you must pass an instance of `IloEnv` as a parameter to the constructor of an extractable object. The environment manages services for the extractable object, such services as memory management.

Extractable Objects and Memory Management

Classes of extractable objects are handle classes. Consequently, the `end` member function must be called to delete an extractable object. The member function `IloEnv::end` calls the member function `IloExtractable::end` for every extractable object created within the invoking environment. For that reason, when you call `IloEnv::end` to clean up the environment, it will automatically delete the extractable objects that you created within that environment as well, making it unnecessary for you to call the `end` member function of each extractable object explicitly yourself.

Extractable Objects and Names

There are two ways for you to name an extractable object. You can pass a name (a C++ string) as a parameter to the constructor when you create an extractable object, or you can call the member function `setName` on the invoking extractable object in order to name it later, after its creation. The member function `getName` accesses the name of an extractable

object; when it returns 0 (zero) there is no name currently associated with the invoking object.

Extractable Objects and IloAny

As we hinted earlier, you can associate your own object with an extractable object. For example, you might want to put user-defined data into an object and then associate that object with your extractable object. You do so by means of the Concert Technology type `IloAny`, defined as `void*`. In other words, you can associate a pointer to any kind of object with an extractable object. In the following line, `e` indicates an extractable object, that is, an instance of `IloExtractable` or one of its derived subclasses, and `usrptr` is an instance of `IloAny`, that is, a pointer to your own user object to associate with the extractable.

```
e.setObject(usrptr);
```

After you have associated your own object with an extractable in that way, you can then use the member function `getObject` to access your object again.

Models: IloModel

When we introduced extractable objects, we remarked that Concert Technology enables you to distinguish the data structures of your problem from the model of your problem and from the algorithm to solve your problem. In that schema, you use an instance of the class `IloModel` to represent the model of your problem. In one sense, a model is a collection of extractable objects. You can add more extractable objects to a model by means of its member function `add` or by means of the template function `IloAdd`. (For examples of how to add extractable objects to a model, see *Adding Objects to a Model*.) You can also remove extractable objects from a model by means of its member function `remove`.

Models are extractable objects themselves. That is, the class `IloModel` derives from `IloExtractable`, as you see in Figure 3.1. This fact allows you to structure models hierarchically if need be. For example, if you have a model defining a core problem and several other models defining different scenarios with various other constraints, you can represent these possibilities in a hierarchic model in pseudo-code like this:

```
IloModel core(env);
core.add(...);
IloModel scenario1(env);
IloModel scenario2(env);
scenario1.add(core);
scenario1.add(...);
secnario2.add(core);
secnario2.add(...);
```

Models are centrally important in Concert Technology as they are the means for passing information to algorithms (that is, the instances of subclasses of `IloAlgorithm`) for solving your problem.

Variables

In a Concert Technology model, you represent the numeric variables of your problem as instances of `IloNumVar`, documented in the *ILOG Concert Technology Reference Manual*. `IloNumVar` provides facilities for defining floating-point variables, integer variables, and Boolean variables (also known as binary or 0-1 variables).

Concert Technology also offers semi-continuous variables represented by instances of the class `IloSemiContVar`, documented in the *ILOG Concert Technology Reference Manual.*

Concert Technology also provides classes of arrays, such as `IloNumVarArray` and `IloSemiContVarArray`, with special constructors and operators to facilitate your definition of a model to suit your problem.

Besides the documentation of these classes in the *ILOG Concert Technology Reference Manual*, you will also find examples in this manual and in the user's manuals of ILOG optimization products, showing you how to use these classes.

Expressions

In Concert Technology, values may be combined with variables to form *expressions*. The values may be numeric values (that is, `IloNum`), whether floating-point, integer, or Boolean. The variables may be:

- numeric (that is, instances of `IloNumVar`);
- semi-continuous (instances of `IloSemiContVar`);
- linear (instances of `IloPiecewiseLinear`);
- constraints (instances of `IloConstraint` and its derived subclasses).

To combine the values with variables to form an expression, you can use the usual arithmetic operators, such as addition +, subtraction -, multiplication *, and division /. The expressions you build may be linear or non linear.

Instances of the class `IloExpr` represent expressions in Concert Technology. You can use expressions themselves in turn to build more complicated expressions. The class `IloExpr`, for example, offers operators such as `operator +=`, `-=`, `*=`, and `/=` for that purpose. Those operators may be useful to you in building range constraints or columns in a model, for example. In the documentation of the class `IloExpr` in the *ILOG Concert Technology*

Reference Manual, you will find examples and programming hints about how to use these operators effectively.

You will also find there an explanation of assumptions about linearity in expressions, as well as documentation of the nested class `IloExpr::LinearIterator` for traversing the terms in the linear part of an expression.

Normalizing Linear Expressions or Reducing Linear Terms

Normalizing is sometimes known as *reducing the terms* of a linear expression.

Linear expressions consist of terms made up of constants and variables related by arithmetic operations; for example, x + 3y is a linear expression of two terms consisting of two variables. In some expressions, a given variable may appear in more than one term, for example, x + 3y +2x. Concert Technology has more than one way of dealing with linear expressions in this respect, and you control which way Concert Technology treats expressions from your application.

In one mode, Concert Technology analyzes linear expressions that your application passes it and attempts to reduce them so that a given variable appears in only one term in the linear expression. This is the default mode. You set this mode with the member function `IloEnv::setNormalizer(IloTrue)`, documented in the *Concert Technology Reference Manual*.

In the other mode, Concert Technology assumes that no variable appears in more than one term in any of the linear expressions that your application passes to Concert Technology. We call this mode *assume normalized linear expressions*. You set this mode with the member function `IloEnv::sNormalizer(IloFalse)`, documented in the *Concert Technology Reference Manual* .

In classes such as `IloExpr` or `IloRange`, there are member functions that check the setting of the member function `IloEnv::setNormalizer` in the environment and behave accordingly. The documentation of those member functions indicates how they behave with respect to normalization.

When you set `IloEnv::setNormalizer(IloFalse)`, those member functions assume that no variable appears in more than one term in a linear expression. This mode may save time during computation, but it entails the risk that a linear expression may contain one or more variables, each of which appears in one or more terms. Such a case may cause certain assertions in member functions of a class to fail if you do not compile with the flag `-DNDEBUG`.

By default, (that is, when `IloEnv::setNormalizer(IloTrue);`) those member functions attempt to reduce expressions. This mode may require more time during preliminary computation, but it avoids of the possibility of a failed assertion in case of duplicates.

Trigonometric, Algebraic, and Symbolic Expressions

Concert Technology also offers such functions as the trigonometric functions `IloSin`, `IloCos`, `IloTan`, plus their arc-equivalents, and algebraic functions, such as `IloExponent`, `IloLog`, `IloPower`, `IloSquare`, and `IloAbs` (for absolute value), for the same purpose of building more complicated expressions in your models. These functions, too, are documented in the *ILOG Concert Technology Reference Manual*.

Algorithms in Concert Technology

In *A Concert Technology Application*, we outline the steps in a basic application using Concert Technology. This chapter has thus far offered a bit more explanation about these first steps:

1. Create an environment.
2. Assemble the data for the problem.
3. Build a model.

Now we are ready to consider the next steps in greater detail:

4. Extract the model for an algorithm.
5. Solve.
6. Access the solution.

In Concert Technology, an algorithm is an object, an instance of a subclass of `IloAlgorithm`, such as `IloCplex`, documented in the *ILOG CPLEX Reference Manual*, for mixed integer programming (MIP) or `IloSolver`, documented in the *ILOG Solver Reference Manual*, for constraint programming.

Extensions of `IloSolver` are available in other optimization products, such as ILOG Scheduler, Dispatcher, and Configurator, each documented in its own reference and user's manual.

As you see in *Extracting a Model for an Algorithm*, you create an algorithm by calling the constructor of its class with an environment, an instance of `IloEnv`.

Algorithms and Memory Management

As a C++ object, an algorithm is a handle. To delete an algorithm, its `end` member function must be called. When you call `IloEnv::end` to clean up the invoking environment, that member function calls the `end` member function of any algorithm created with that invoking

environment as a parameter. In other words, ending the environment automatically ends any algorithm created in that environment.

Algorithms and Extraction

After you create an algorithm, normally in your Concert Technology application, your next step is to extract a model for that algorithm. As you see in *Extracting a Model for an Algorithm*, you call the member function extract with the model as an argument. Since these two steps—creating the algorithm and extracting the model—are repeated in most Concert Technology applications, there is a shortcut combining them, like this:

```
IloCplex algo(model);
```

When an algorithm extracts a model, it scans all the extractable objects in that model and creates corresponding internal objects for optimizing the model.

To help you understand exactly what is extracted from a model, you might think of a model as a graph of nodes and arcs. The extractable objects explicit in the model are represented in that graph as nodes. Whenever an extractable object in that graph keeps a reference to any other extractable object, that reference is represented in our graph by an arc to a node representing that referenced extractable object. For example, a range constraint (that is, an instance of `IloRange` in a model) references all the variables used in any expressions to build up the range, so the range appears in our graph as a node with arcs connecting it to the nodes presenting those variables.

When an algorithm extracts an extractable object, it recursively extracts the other extractable objects referenced by that extractable object. To return to our image of a graph of nodes and arcs, when an algorithm extracts an object, it also extracts the subgraph rooted at that object. Theoretically, an algorithm traversing its extraction graph may reach the same node more than once, but practically, the algorithm will extract only one copy of the extractable object.

An algorithm extracts only one model at a time. If you attempt to extract another model for an instance of `IloCplex`, that algorithm will discard the first model and then extract the new one. If you extract another model for an instance of `IloSolver`, that algorithm will attempt to synchronize (or not synchronize) changes between the first and second model, according to parameters that you pass.

Not all algorithms can extract all extractable objects. For example, an instance of `IloCplex` cannot deal with range constraints involving non linear terms. In such a situation, an extraction may fail, and the member function `extract` will throw an exception containing a list of the extractable objects that it failed to extract.

Solving to Get Results

After you create an algorithm and extract a model for it, you simply call the member function `solve`, as you see in *Solving the Problem* and in each of the examples treated in later chapters of this manual.

That member function returns a Boolean value, `IloTrue` indicating that a feasible solution has been found. You can access that feasible solution through member functions of the algorithm, such as `getStatus` and `getValue`. While such a solution may be feasible, in an instance of `IloCplex`, it may not be optimal.

When the `solve` member function returns `IloFalse`, it means that no feasible solution was generated during that particular invocation of the algorithm. This return value does not necessarily mean that there is no solution nor that the model is infeasible. For example, the model may include time limits that stopped the search for a solution before an optimal solution could be found.

The *ILOG Concert Technology Reference Manual* documents the return values of `getStatus`. The *ILOG CPLEX User's Manual* explains how to interpret those values with respect to `IloCplex`.

Modification of Models

So far, we have explained how to develop a Concert Technology application based on a sole model. In fact, there may be situations in which you want to develop a series of models, for example, different models representing various scenarios, and Concert Technology supports that kind of modeling as well. You can create an initial model and then modify it repeatedly.

You can modify a model even after it has been extracted to an algorithm. Not only can you modify an extracted model, but whenever you do so, the modification is tracked by the algorithm through the notification system in Concert Technology. When an extractable object is modified by means of Concert Technology member functions, those member functions notify all algorithms that have extracted that object, and the algorithms behave appropriately changing their internal representation of the extracted object according to the modification.

Member functions that modify extractable objects and notify algorithms that have extracted the object are clearly marked in the *ILOG Concert Technology Reference Manual*, like this:

> ***Note:*** *This member function notifies Concert Technology algorithms about this change of this invoking object.*

Conversion of a Variable

When you want to change the type of a numeric variable, for example, from floating-point to integer, or from integer back to floating-point, you do so by adding an instance of `IloConversion` to the model. This technique makes it possible for you to use the same variable with different types (floating-point, integer, and so forth) in different models.

As an illustration, let's assume that we have an integer variable, `var`. When an algorithm, such as an instance of `IloCplex`, extracts `var`, it creates an internal representation of the variable as integer. If we want to change the type of `var` from integer to floating-point, for example, then we create an instance of `IloConversion`, and we add that conversion to the model. When `IloCplex` extracts the conversion, it changes its own internal representation of `var` to reflect this change in its type. The variable `var` itself is *not* affected by this change; only its internal representation in the algorithm is converted. If we remove the conversion from the model, then `IloCplex` will restore its original representation of `var` as an integer variable.

In this way, we have the possibility of using the same variable with different types in different models. That is, the variable may be integer in one model, floating-point in another. In relaxations of MIPs, for example, this convention may be useful.

The following lines, for example, first create a MIP model (`mip`) followed by the creation of its relaxation (`relax`).

```
IloModel mip(env);
IloModel relax(env);
relax.add(mip);
relax.add (IloConversion(env, vars, ILOFLOAT));
```

In other words, the model `relax` contains the same variables as the model `mip`, but in `relax` the variables have been relaxed to floating-point.

Timers

An instance of the class `IloTimer` works like a stop watch in a Concert Technology application. Member functions of this class enable you to set the timer going, reset it when you want, or restart it at appropriate points in your application. The member function `IloTimer::getTime` tells you the amount of elapsed time in seconds. For example, you can use repeated calls to `getTime` in order to measure the amount of time spent on various parts of the program. You might then store the results of those calls as elements in an array, `Times`, and use those elements in reports to indicate how much time passed in key parts of the application.

```
IloTimer timer;
double Times[3];
timer.start();
```

There are timers associated with environments (instances of `IloEnv`) and with algorithms (instances of the subclasses of `IloAlgorithm`). The timer associated with an environment measure time since the creation of the environment.

```
env.getTime();
```

In contrast, the timer associated with an algorithm measure the time spent during the execution of the member function `IloAlgorithm::solve`. When you construct an algorithm, its timer is reset. At the beginning of the execution of `IloAlgorithm::solve`, the timer starts, and it stops when the optimization terminates. The following call returns the accumulated time spent in all calls of `solve`.

```
algorithm.getTime();
```

Error Handling

To help you avoid or eliminate programming bugs in your Concert Technology application, the library uses `assert` statements extensively. For example, every member function of a handle class first asserts that the handle is not null. Likewise, member functions that accept an array as a parameter assert that length of the array is consistent. This use of `assert` is practical for avoiding certain kinds of programming errors, because once you as an application developer have eliminated all the bugs from your program, you probably don't want to spend any run time in a deployment version of your application to verify the absence of such anomalies as these `assert` statements can detect. Following the C++ standard, the assertions in Concert Technology are included in the code by default. They can be turned off when an application is compiled (for example, when it is compiled for deployment) with the compiler option `-DNDEBUG`.

There are other anomalies, however, that cannot be detected and avoided by a judicious use of assertions. Such anomalies must be detected in deployed applications, so that your application can respond appropriately. For that purpose in Concert Technology we have used C++ exceptions. All the exceptions that may potentially be thrown by member functions and global functions of Concert Technology are instances of `IloException` or its derived subclasses, documented in the *ILOG Concert Technology Reference Manual*.

In order for your application to respond gracefully to such exceptions, we urge you to encapsulate your Concert Technology code within `try` and `catch` statements, like this:

```
IloEnv env;
try {
  // Concert Technology code here
}
catch (IloException& e) {
  // handle the Concert Technology exception
}
catch (...) {
  // handle any other exception
}
env.end();
```

The environment in those lines is declared outside the `try` block, because even in the case of an exception, the member function `IloEnv::end` must be called. To handle potential

failures in the creation of the environment itself, we recommend that you add another pair of `try` and `catch` statements.

We catch the exception by reference (rather than by value) so that we preserve any potential subclass known when the exception is caught. For example, when the following line prints the exception, this practice of catching by reference may preserve more type information than a catch by value.

```
cerr << e << endl;
```

Naturally, we recommend that every exception should be caught, and in fact, we recommend that you should catch exceptions by reference.

C H A P T E R

4

Transport: Piecewise Linear Optimization

This chapter shows you how to represent piecewise linear optimization in Concert Technology, whether you are using an instance of `IloCplex`, `IloSolver`, or `IloLinConstraint`. In this chapter, you will learn:

- how to use classes of Concert Technology to represent piecewise linear functions;
- how to determine whether branching is necessary in a piecewise linear problem.

Piecewise Linearity in Concert Technology

Some problems are most naturally represented by constraints over functions that are not purely linear but consist of linear *segments*. Such functions are sometimes known as piecewise linear. In this chapter, we use a transportation example to show you various ways of stating and solving problems that lend themselves to a piecewise linear model. Before we plunge into the problem itself, we define a few terms that we rely on in the remainder of this discussion.

What Is a Piecewise Linear Function?

From a geometric point of view, Figure 4.1 shows a conventional piecewise linear function `f(x)`. This particular function consists of four segments. If we consider the function over four separate intervals, `(-∞, 4)` and `[4, 5)` and `[5, 7)` and `[7, ∞)`, we see that `f(x)` is

linear in each of those separate intervals. For that reason, we say it is *piecewise linear.* Within each of those segments, the slope of the linear function is clearly increasing or decreasing. The points where the slope of the function changes are known as *breakpoints*. The piecewise linear function in Figure 4.1 has three breakpoints.

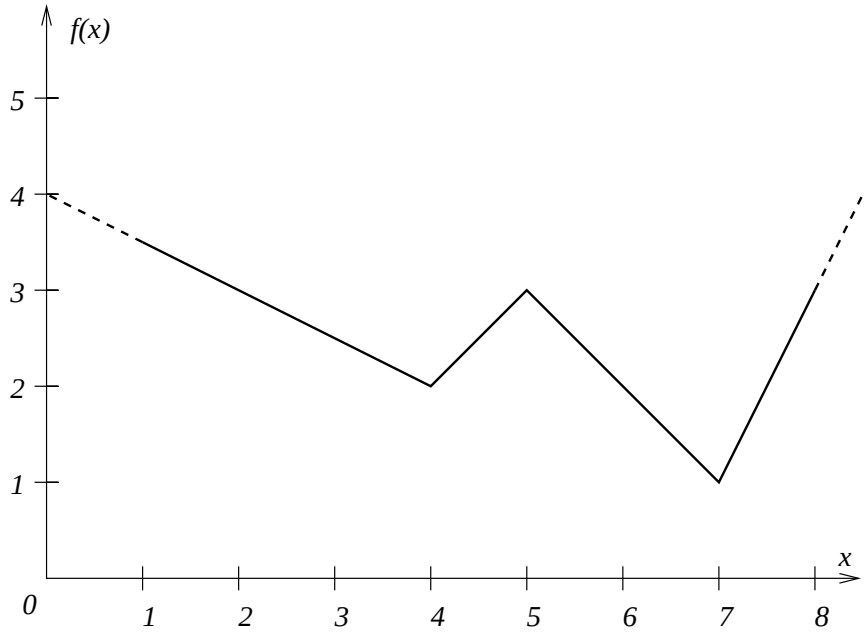

Figure 4.1 *A piecewise linear function with breakpoints*

Piecewise linear functions are often used to represent or to approximate non linear unary functions (that is, non linear functions of one variable). For example, piecewise linear functions frequently represent situations where costs vary with respect to quantity or gains vary over time.

Syntax of Piecewise Linear Functions

To define a piecewise linear function in ILOG Concert Technology, we need these components:

- the variable of the piecewise linear function;
- the breakpoints of the piecewise linear function;
- the slope of each segment (that is, the rate of increase or decrease of the function between two breakpoints);
- the geometric coordinates of at least one point of the function.

In other words, for a piecewise linear function of n breakpoints, we need to know n+1 slopes.

Typically, we specify the breakpoints of a piecewise linear function as an array of numeric values. For example, we specify the breakpoints of our function f(x) as it appears in Figure 4.1 in this way:

```
IloNumArray (env, 3, 4., 5., 7)
```

We indicate the slopes of its segments as an array of numeric values as well. For example, we specify the slopes of our example in this way:

```
IloNumArray (env, 4, -0.5, 1., -1., 2.)
```

And we need to specify the geometric coordinates of at least one point of the function, (x, f(x)); for example, (4, 2). Then in ILOG Concert Technology, we pull together those details in an instance of the class IloPiecewiseLinear in this way:

```
IloPiecewiseLinear(x,
                   IloNumArray(env, 3, 4., 5., 7.),
                   IloNumArray(env, 4, -0.5, 1., -1., 2.),
                   4, 2)
```

Discontinuous Piecewise Linear Functions

Thus far, we have been talking about a piecewise linear function where the segments are *continuous*. Intuitively, in a continuous piecewise linear function, the endpoint of one segment has the same coordinates as the initial point of the next segment, as in Figure 4.1.

There are piecewise linear functions, however, where two consecutive breakpoints may have the same x coordinate but differ in the value of f(x). We refer to such a difference as a *step* in the piecewise linear function, and we say that such a function is *discontinuous*. Figure 4.2 shows a discontinuous piecewise linear function with two steps.

Syntactically, we represent a step in this way:

- The value of the first point of a step in the array of slopes is the *height* of the step.
- The value of the second point of the step in the array of slopes is the *slope* of the function after the step.

By convention, we say that a breakpoint belongs to the segment that starts at that breakpoint. For example, in Figure 4.2, f(3) is equal to 3 (not 1); similarly, f(5) is equal to 5.

In ILOG Concert Technology, we represent a discontinuous piecewise linear function as an instance of the class `IloPiecewiseLinear` (the same class that we use for continuous piecewise linear functions). For example, we declare the function in Figure 4.2 in this way:

```
IloPiecewiseLinear(x,
                   IloNumArray(env, 4, 3. ,3. ,5. ,5.),
                   IloNumArray(env, 5, 0., 2., 0.5, 1., -1.),
                   0, 1);
```

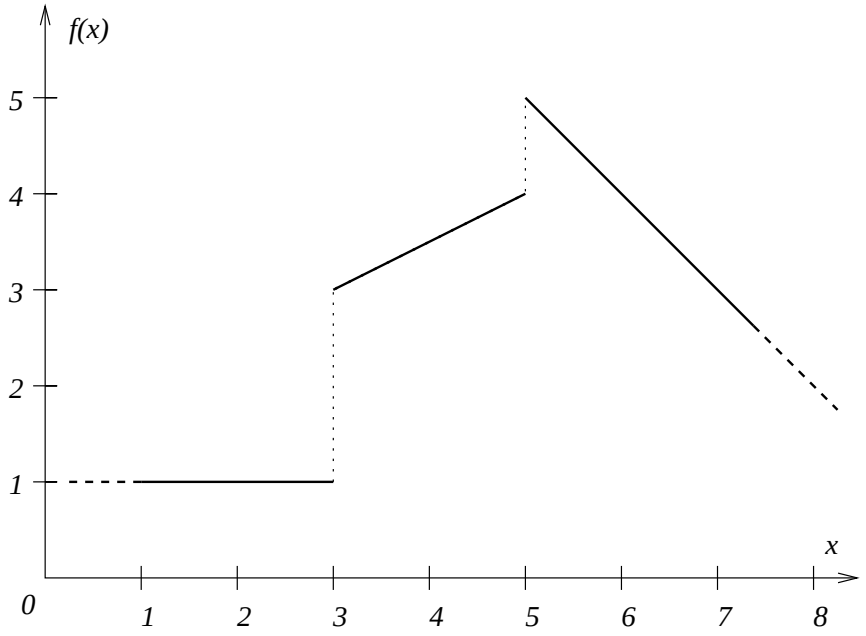

Figure 4.2 *A discontinuous piecewise linear function with steps*

Using IloPiecewiseLinear

Whether it represents a continuous or a discontinuous piecewise linear function, an instance of `IloPiecewiseLinear` behaves like a floating-point *expression*. That is, you may use it in a term of a linear expression or in a constraint added to a model (an instance of `IloModel`).

If you are licensed to use `IloCplex`, then you can use instances of `IloPiecewiseLinear` in models extracted for that algorithm. *Solving the Problem* later in this chapter offers more detail about this possibility.

If you are licensed to use `IloLinConstraint`, then you can also use instances of `IloPiecewiseLinear` in a linear program extracted by an instance of `IloLinConstraint`.

When a piecewise linear expression over a variable `x` appears in a constraint added to an instance of `IloSolver` (documented in the *ILOG Solver Reference Manual*), the solver performs constraint propagation and domain reduction as usual for expressions. More precisely, `IloSolver` maintains the smallest interval for the expression with regard to the domain of `x`. For example, if `x` lies between `5` and `7`, the expression represented by the piecewise linear function in Figure 4.1 lies between `1` and `3`. Conversely, the domain imposed on the expression reduces the domain of `x`. For example, if the domain of the expression lies between `1` and `1.5`, the domain of `x` lies between `6.5` and `7.25`.

If you add an instance of `IloPiecewiseLinear` to an instance of `IloLinConstraint` (documented in the *ILOG Hybrid Optimizers User's Guide & Reference*), then the linear program extracted by `IloLinConstraint` automatically treats the expression represented by `IloPiecewiseLinear` appropriately. It linearizes the piecewise linear expression to obtain a tight relaxation of the expression by adding variables and internal constraints as needed. Roughly, for an expression having `n` slopes, about `n` variables are introduced as well as three constraints.

Adding a piecewise linear expression to an instance of `IloSolver` in addition to adding it to an instance of `IloLinConstraint` can help reduce the search space. The reasoning about bounds carried out by `IloSolver` excludes unreachable segments, while the global view of `IloLinConstraint` detects infeasibilities earlier. We will see this behavior in the example we walk through in *Describing the Problem* later in this chapter.

Special Considerations about Solving with IloPiecewiseLinear

In the general case, a problem containing piecewise linear constraints is equivalent to a mixed-integer programming problem (MIP). Consequently, a branching strategy is necessary to solve it. A possible way to branch on a piecewise linear function is to try the segments of the function exhaustively to obtain subproblems where the function is linear. In the example of Figure 4.1, this strategy entails setting a choice point with four choices by setting the bounds of `x` to `(-∞, 4)` or `[4, 5)` or `[5, 7)` or `[7, ∞)`.

A better approach is a binary search (also known as dichotomizing the search) that sets a choice point with two choices such that the number of segments that remain in each alternative is almost the same. In the example in Figure 4.1, this strategy sets the choice point `x <= 5` or `x >= 5`.

Under some conditions, a piecewise linear problem may be equivalent to a linear program (rather than a MIP). In such a case, no branching is necessary because `IloLinConstraint` provides a simple way to obtain a solution. Whether a problem represented by a piecewise linear function made up of segments can be modeled as a linear program (and thus we can solve it straightforwardly without branching) depends on the slope of these segments.

A piecewise linear function is *convex* if the succession of the slopes of its segments is increasing. In contrast, we say that a piecewise linear function is *concave* if the succession of slopes is decreasing.

Consider a minimization problem having linear constraints of the form

$a_1x_1+...+a_nx_n <= a_0$ and

$a_1x_1+...+a_nx_n = a_0$ where x_i is a variable or a piecewise linear function.

Sufficient conditions for this problem to be a linear program are these:

- every piecewise linear function is convex or concave;
- every convex function appears with a positive coefficient in <= constraints or in the objective function;
- every concave function appears with a negative coefficient in <= constraints or in the objective function.

(These statements have equivalent translations for >= constraints or for a maximization problem.) With this background about piecewise linear functions and their representation in ILOG Concert Technology, let's consider a transportation example exploiting piecewise linearity.

Describing the Problem

Let's assume that a company must ship cars from factories to showrooms. Each factory can supply a fixed number of cars, and each showroom needs a fixed number of cars. There is a cost for shipping a car from a given factory to a given showroom. The objective is to minimize the total shipping cost while satisfying the demands and respecting supply.

In concrete terms, let's assume there are three factories and four showrooms. Here is the quantity that each factory can supply:

$supply_0 = 1000$
$supply_1 = 850$
$supply_2 = 1250$

Each showroom has a fixed demand:

$demand_0 = 900$
$demand_1 = 1200$
$demand_2 = 600$
$demand_3 = 400$

We let `nbSupply` be the number of factories and `nbDemand` be the number of showrooms. Let x_{ij} be the number of cars shipped from factory `i` to showroom `j`. The model is

composed of `nbDemand` + `nbSupply` constraints that force all demands to be satisfied and all supplies to be shipped. Thus far, a model for our problem looks like this:

$$\text{minimize} \quad \sum_{i=0}^{nbDemand-1} \sum_{j=0}^{nbSupply-1} cost_{ij} \cdot x$$

subject to

$$\sum_{j=0}^{nbSupply-1} x_{ij} = supply_i \qquad i = 0, \ldots, nbDemand-$$

$$\sum_{i=0}^{nbDemand-1} x_{ij} = demand_j \qquad j = 0, \ldots, nbSupply-$$

Variable Shipping Costs

Now we consider the costs of shipping from a given factory to a given showroom. Let's assume that for every pair (factory, showroom) we have different rates varying according to the quantity shipped. To illustrate the difference between convex and concave piecewise linear functions, in fact, we will assume that we have two different tables of rates for shipping cars from factories to showrooms. Our first table of rates looks like this:

- a rate of 120 per car for quantities between 0 and 200;
- a rate of 80 per car for quantities between 200 and 400;
- a rate of 50 per car for quantities higher than 400.

These costs that vary according to quantity define the piecewise linear function represented in Figure 4.3. As you can see, the slopes of the segments of that function are decreasing, so we say that the function is concave.

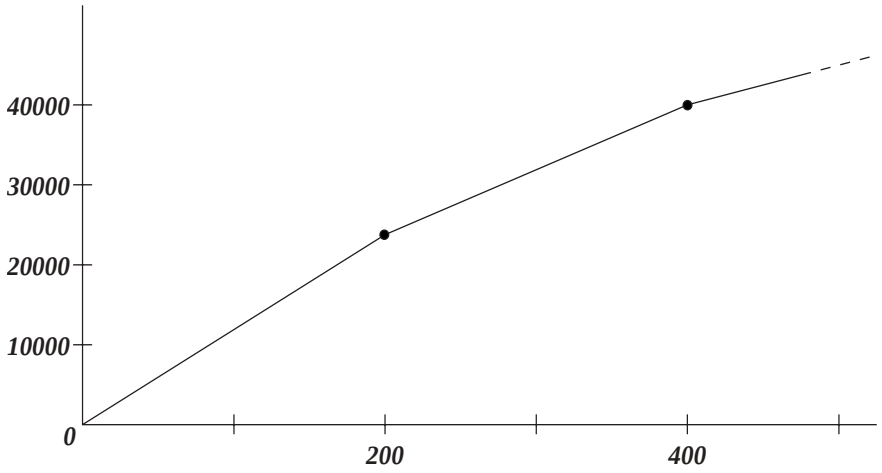

Figure 4.3 *A concave piecewise linear cost function*

Let's also assume that we have a second table of rates for shipping cars from factories to showrooms. Our second table of rates looks like this:

- a rate of 30 per car for quantities between 0 and 200;
- a rate of 80 per car for quantities between 200 and 400;
- a rate of 130 per car for quantities higher than 400.

The costs in this second table of rates that vary according to the quantity of cars shipped define a piecewise linear function, too. It appears in Figure 4.4. The slopes of the segments in this second piecewise linear function are increasing, so we say this function is convex.

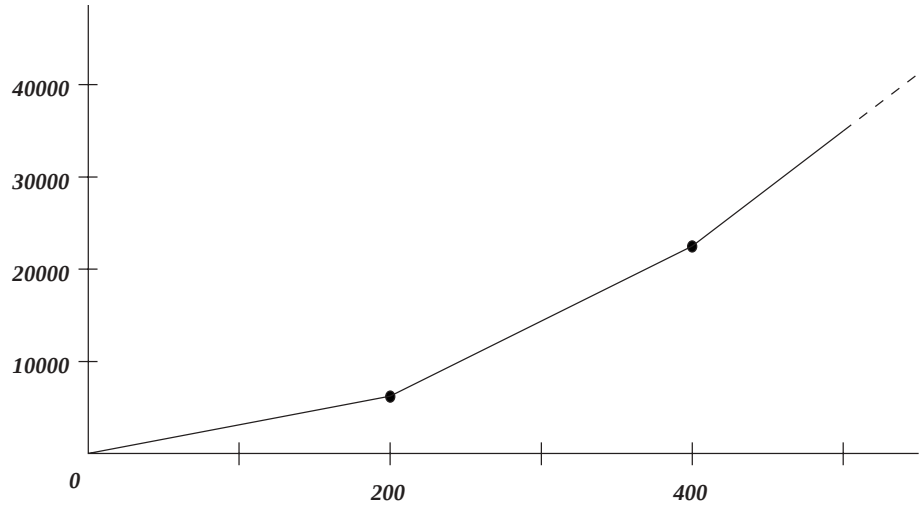

Figure 4.4 *A convex piecewise linear cost function*

Model with Varying Costs

With this additional consideration about costs varying according to quantity, our model now looks like this:

$$\text{minimize} \quad \sum_{i=0}^{nbDemand-1} \sum_{j=0}^{nbSupply-1} y_{ij}$$

subject to

$y_{ij} = f(x_{ij})$ for $i = 0, ..., nbDemand\text{-}1$ and $j = 0, ..., nbSupply\text{-}1$

$$\sum_{j=0}^{nbSupply-1} x_{ij} = demand_i \quad \text{for} \quad i = 0, \ldots, nbDemand-1$$

$$\sum_{i=0}^{nbDemand-1} x_{ij} = supply_j \quad \text{for} \quad j = 0, \ldots, nbSupply-$$

With this problem in mind, let's consider how to represent the data and model in ILOG Concert Technology.

Representing the Data

As we have done in other examples, we will use the ILOG Concert Technology template class `IloArray` in a type definition to create matrices for our problem, like this:

```
typedef IloArray<IloNumArray>    NumMatrix;
typedef IloArray<IloNumVarArray> NumVarMatrix;
```

Those two-dimensional arrays (that is, arrays of arrays) are now available to us to represent the demands from the showrooms and the supplies available from the factories.

```
IloInt nbDemand = 4;
IloInt nbSupply = 3;
IloNumArray supply(env, nbSupply, 1000., 850., 1250.);
IloNumArray demand(env, nbDemand, 900., 1200., 600., 400.);

NumVarMatrix x(env, nbSupply);
NumVarMatrix y(env, nbSupply);
```

Developing a Model

As we have done in other examples in this manual, we begin by creating an environment, an instance of `IloEnv`, documented in the *ILOG Concert Technology Reference Manual*.

```
IloEnv env;
```

Within that environment, we create a model for our problem, an instance of `IloModel`, also documented in the *ILOG Concert Technology Reference Manual.*

```
IloModel model(env);
```

Then we add constraints and an objective to our model.

Adding Constraints

According to the description of our problem, we want to be sure that the supply of cars from the factories meets the demand of the showrooms. At the same time, we want to be sure that we do not ship cars that are not in demand; in terms of our model, the demand should meet

the supply as well. We represent those ideas as constraints that we add to our model, like this:

```
for(i = 0; i < nbSupply; i++){
  x[i] = IloNumVarArray(env, nbDemand, 0, IloInfinity, ILOFLOAT);
  y[i] = IloNumVarArray(env, nbDemand, 0, IloInfinity, ILOFLOAT);
}

for(i = 0; i < nbSupply; i++){      // supply must meet demand
  model.add(IloSum(x[i]) == supply[i]);
}
for(j = 0; j < nbDemand; j++){      // demand must meet supply
  IloExpr v(env);
  try {
  for(i = 0; i < nbSupply; i++)
    v += x[i][j];
  model.add(v == demand[j]);
}
catch(...) {
    v.end();
    throw;
  }
  v.end();
}
```

Programming Practices

As a good programming practice, we enclose the loop that uses the temporary expression `v` between `try` and `catch`.

Optionally, we also delete the temporary expression `v` by calling the member function `IloExpr::end` when the expression is no longer needed. In strictest terms, that clean up is not necessary because the expression `v` has been created explicitly in the environment `env`, so it will be deleted automatically when we call `env.end`. However, in applications where resources are so limited that reclaiming memory allocated to temporary expressions like this one may be advantageous, this optional clean up is available to you.

Checking Convexity and Concavity

To illustrate the ideas of convex and concave piecewise linear functions, we introduced in our problem description two tables of costs that vary according to the quantity of cars shipped. To accommodate those two tables in our model, we add the following lines.

```
if (convex) {
  for(i = 0; i < nbSupply; i++){
    for(j = 0; j < nbDemand; j++){
      model.add(y[i][j] == IloPiecewiseLinear(x[i][j],
                                        IloNumArray(env, 2, 200., 400.),
                                        IloNumArray(env, 3, 30., 80., 130.),
                                        0, 0));
    }
  }
}else{
  for(i = 0; i < nbSupply; i++){
    for(j = 0; j < nbDemand; j++){
      model.add(y[i][j] == IloPiecewiseLinear(x[i][j],
                                        IloNumArray(env, 2, 200., 400.),
                                        IloNumArray(env, 3, 120., 80., 50.),
                                        0, 0));
    }
  }
}
```

Adding an Objective

We want to minimize costs of supplying cars from factories to showrooms. To represent that objective, we add these lines:

```
IloExpr obj(env);
for(i = 0; i < nbSupply; i++){
  obj += IloSum(y[i]);
}

model.add(IloMinimize(env, obj));
```

Solving the Problem

If our problem satisfies the special conditions of a simple linear program, as we outlined in *Special Considerations about Solving with IloPiecewiseLinear*, then an instance of `IloCplex` (documented in the *ILOG CPLEX Reference Manual*) will solve the problem with a simplex linear optimizer.

On the other hand, if our problem fits the more general case, and thus requires a mixed integer programming (MIP) for its solution, an instance of `IloCplex` will solve the problem with a MIP optimizer using branch & bound techniques.

In either case, in our program we need only the following few lines. They create an algorithm (an instance of `IloCplex` documented in the *ILOG CPLEX Reference Manual*) in an environment (an instance of `IloEnv` documented in the *ILOG Concert Technology Reference Manual*) and extract the model (an instance of `IloModel` also documented in the *ILOG Concert Technology Reference Manual*) for that algorithm to find a solution.

```
IloCplex cplex(env);
cplex.extract(model);
cplex.solve();
```

Displaying a Solution

To display the solution, we use the member functions of `IloEnv` (documented in the *ILOG Concert Technology Reference Manual*) and `IloCplex` (documented in the *ILOG CPLEX Reference Manual*).

```
env.out() << " - Solution: " << endl;
for(i = 0; i < nbSupply; i++){
  env.out() << "   " << i << ": ";
  for(j = 0; j < nbDemand; j++){
    env.out() << cplex.getValue(x[i][j]) << "\t";
  }
  env.out() << endl;
}
env.out() << "   Cost = " << cplex.getObjValue() << endl;
```

Ending the Application

As we have done in other examples in this manual, we end our application with a call to the member function `IloEnv::end` to clean up the memory allocated for the environment and algorithm.

```
env.end();
```

Complete Program

You can see the complete program here or on line in the standard distribution of `IloCplex` in the file `/examples/src/transport.cpp`. To run this example, you need a license for ILOG CPLEX.

```
// ------------------------------------------------------------ -*- C++ -*-
// File: examples/src/transport.cpp
// --------------------------------------------------------------------------
//  Copyright (C) 1999-2001 by ILOG.
//  All Rights Reserved.
//  Permission is expressly granted to use this example in the
//  course of developing applications that use ILOG products.
// --------------------------------------------------------------------------
//

#include <ilcplex/ilocplex.h>

ILOSTLBEGIN

typedef IloArray<IloNumArray>    NumMatrix;
typedef IloArray<IloNumVarArray> NumVarMatrix;

int main(int argc, char** argv) {
  if (argc <= 1) {
     cerr << "Usage: " << argv[0] << " <model>" << endl;
     cerr << "  model = 0 -> convex piecewise linear model, " << endl;
     cerr << "  model = 1 -> concave piecewise linear model. [default]"
                                                              << endl;
  }
  IloBool convex;
  if (argc <= 1)
    convex = IloFalse;
  else
    convex = atoi(argv[1]) == 0 ? IloTrue : IloFalse;

  IloEnv env;
  try {
    IloInt i, j;
    IloModel model(env);

    IloInt nbDemand = 4;
    IloInt nbSupply = 3;
    IloNumArray supply(env, nbSupply, 1000., 850., 1250.);
    IloNumArray demand(env, nbDemand, 900., 1200., 600., 400.);

    NumVarMatrix x(env, nbSupply);
    NumVarMatrix y(env, nbSupply);

    for(i = 0; i < nbSupply; i++) {
      x[i] = IloNumVarArray(env, nbDemand, 0, IloInfinity, ILOFLOAT);
```

```
  y[i] = IloNumVarArray(env, nbDemand, 0, IloInfinity, ILOFLOAT);
}

for(i = 0; i < nbSupply; i++) {      // supply must meet demand
  model.add(IloSum(x[i]) == supply[i]);
}
for(j = 0; j < nbDemand; j++) {      // demand must meet supply
  IloExpr v(env);
  try {
    for(i = 0; i < nbSupply; i++)
      v += x[i][j];
    model.add(v == demand[j]);
  }
  catch(...) {
    v.end();
    throw;
  }
  v.end();
}
if (convex) {
  for(i = 0; i < nbSupply; i++) {
    for(j = 0; j < nbDemand; j++) {
      model.add(y[i][j] == IloPiecewiseLinear(x[i][j],
                                    IloNumArray(env, 2, 200., 400.),
                                    IloNumArray(env, 3, 30., 80., 130.),
                                    0, 0));
    }
  }
} else {
  for(i = 0; i < nbSupply; i++) {
    for(j = 0; j < nbDemand; j++) {
      model.add(y[i][j] == IloPiecewiseLinear(x[i][j],
                                    IloNumArray(env, 2, 200., 400.),
                                    IloNumArray(env, 3, 120., 80., 50.),
                                    0, 0));
    }
  }
}

IloExpr obj(env);
for(i = 0; i < nbSupply; i++) {
  obj += IloSum(y[i]);
}
model.add(IloMinimize(env, obj));

IloCplex cplex(env);
cplex.extract(model);
cplex.solve();

env.out() << " - Solution: " << endl;
for(i = 0; i < nbSupply; i++) {
  env.out() << "   " << i << ": ";
  for(j = 0; j < nbDemand; j++) {
```

```
            env.out() << cplex.getValue(x[i][j]) << "\t";
          }
          env.out() << endl;
        }
        env.out() << "   Cost = " << cplex.getObjValue() << endl;
      } catch(IloException& e) {
        cerr << "ERROR: " << e.getMessage() << endl;
      } catch (...) {
        cerr << "Error" << endl;
      }
      env.end();
      return 0;
    }
```

Part II

Meet the Players

This part of the manual introduces the other products from ILOG that enrich your ILOG Concert Technology applications. For each of those products, this part includes an example to give you a hint of how that product extends and enhances the facilities you've already encountered in ILOG Concert Technology.

ILOG CPLEX supports mathematical programming with ILOG Concert Technology, including primal and dual simplex optimizers, barrier optimizers, a network simplex optimizer, and a branch & cut optimizer for mixed integer programming.

ILOG Solver provides the facilities for constraint programming with ILOG Concert Technology. It serves as the basis for the other constraint programming products from ILOG, such as ILOG Scheduler, Dispatcher, and Configurator. It also offers the means for extending constraint programming facilities yourself.

ILOG Scheduler extends constraint programming facilities for scheduling and resource allocation problems.

ILOG Dispatcher extends constraint programming facilities for vehicle routing problems and for technician dispatching problems.

ILOG Configurator extends constraint programming facilities for configuration problems.

C H A P T E R

5

Cutting Stock: Column Generation

This chapter uses an example of cutting stock to demonstrate the technique of column generation in Concert Technology. In it, you will learn:

- how to use classes of Concert Technology for column generation in column-wise modeling;
- how to modify a model and re-optimize;
- how to change the type of a variable with `IloConversion`;
- how to use more than one model;
- how to use more than one algorithm (instances of `IloCplex`, for example);
- how to use a timer.

What Is Column Generation?

In colloquial terms, column generation is a way of beginning with a small, manageable *part* of a problem (specifically, a few of the variables), solving that part, analyzing that partial solution to determine the next part of the problem (specifically, one or more variables) to add to the model, and then resolving the enlarged model. Column-wise modeling repeats that process until it achieves a satisfactory solution to the whole of the problem.

In formal terms, column generation is a way of solving a linear programming problem that adds columns (corresponding to constrained variables) during the pricing phase of the simplex method of solving the problem. In gross terms, generating a column in the primal simplex formulation of a linear programming problem corresponds to adding a constraint in its dual formulation. In the dual formulation of a given linear programming problem, you might think of column generation as a cutting plane method.

In that context, many researchers have observed that column generation is a very powerful technique for solving a wide range of industrial problems to optimality or to near optimality. Ford and Fulkerson, for example, suggested column generation in the context of a multi-commodity network flow problem as early as 1958 in the journal of *Management Science*. By 1960, Dantzig and Wolfe had adapted it to linear programming problems with a decomposable structure. Gilmore and Gomory then demonstrated its effectiveness in a cutting stock problem. More recently, vehicle routing, crew scheduling, and other integer-constrained problems have motivated further research into column generation.

Column generation rests on the fact that in the simplex method, we do not need access to all the variables of the problem simultaneously. In fact, we can begin work with only the basis—a particular subset of the constrained variables—and then use reduced cost to determine which other variables we want to access as needed.

Column-Wise Models in Concert Technology

Concert Technology offers facilities for exploiting column generation. In particular, you can design the model of your problem (one or more instances of the class `IloModel`) in terms of a basis and added columns (instances of `IloNumVar`, `IloNumVarArray`, `IloNumColumn`, or `IloNumColumnArray`). For example, instances of `IloNumColumn` represent *columns*, and you can use `operator()` in the classes `IloObjective` and `IloRange` to create *terms* in column expressions. In practice, the column serves as a kind of place holder for a variable in other extractable objects (such as a range constraint or an objective) when your application needs to declare or use those other extractable objects before it can actually know the value of a variable appearing in them.

Furthermore, an instance of `IloCplex` provides a way to solve the master linear problem, while other Concert Technology algorithms (that is, instances of `IlcSolver`, of `IloCplex` itself, or of subclasses of `IloAlgorithm`, for example) lend themselves to other parts of the problem by determining which variables to consider next (and thus which columns to generate).

In the *Concert Technology Reference Manual*, the concept *Column-Wise Modeling* provides more detail about this topic and offers simple examples of its use.

Describing the Problem

The cutting stock problem in this chapter is sometimes known in math programming terms as a knapsack problem with reduced cost in the objective function.

Generally, a cutting stock problem begins with a supply of rolls of material (the stock). Strips are cut from a continuous roll. All the strips cut from one roll are known together as a *pattern*. The point of this example is to use as few rolls of stock as possible to satisfy some specified demand. By convention, we assume that we lay out only one pattern across the stock; consequently, we are concerned with only one dimension—the width—of each roll of stock.

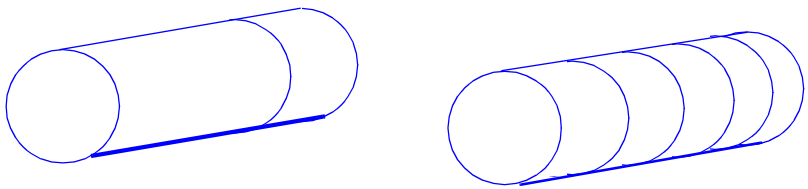

Figure 5.1 *Two different patterns from a roll of stock*

Even with that simplifying assumption, the fact that we can have so many different patterns makes a naive model of this problem (where we declare one variable for every possible pattern) impractical. Such a model introduces too many symmetries. Fortunately, for any given customer order, we can see that a limited number of patterns will suffice, so we can disregard many of the possible patterns and focus on finding the relevant ones.

Here is a conventional statement of a cutting stock problem in terms of the unknown X_j, the number of times that pattern j will be used, and A_{ij}, the number of items i of each pattern j needed to satisfy demand d_i:

Minimize $\sum_{i} X_j$

subject to $\sum_{ii} A_{ij} X_j$ with $X_j \geq 0$

To solve a cutting stock problem by column generation, we first start with a subproblem. We choose one pattern, lay it out on the stock, and cut as many items as possible, subject to the constraints of demand for that item and the width of the stock. This procedure will surely work in that we will get some answer—a feasible solution—to our problem, but we will not necessarily get a satisfactory answer in this way.

To move closer to a satisfactory solution, we can then generate other columns. That is, we choose other decision variables (other X_j) to add to our model. We choose those decision variables on the basis of their favorable reduced cost. In linear programming terms, we choose variables for which the negative reduced cost improves our minimization. In this formulation of the rest of the problem, we use π_i as the dual value of constraint i, like this:

Minimize $1 - \sum_i \pi_i A_i$

subject to $\sum_i W_i A_i \leq W$

where W is the width of a roll of stock and A_i must be a non-negative integer.

Representing the Data

As usual in a Concert Technology application, we create an environment, an instance of `IloEnv`, where we will organize the data and build the model of the problem.

The data defining this problem includes the width of a roll of stock. We read this value from a file and represent it by a numeric value, `rollWidth`. We also read the widths of the ordered rolls from a file and put that data into an array of numeric values, `size`. Finally, we read from a file the number of rolls ordered of each width and put that data into an array of numeric values, `amount`.

Developing the Model: Building and Modifying

In this problem, we first build an initial model `cutOpt`. Later, through its modifications, we effectively build another model `patGen` to generate the new columns.

We declare a first model `cutOpt`, an instance of `IloModel`.

```
IloModel cutOpt (env);
```

We declare an array of numeric variables (an instance of `IloNumVarArray`) `Cut[j]` to represent the decision variables in our model. The index `j` indicates a pattern using the widths we find in the array `size`. Consequently, a given variable `Cut[j]` in our solution will be the number of rolls to cut according to pattern `j`.

As we build a model for this problem, we will have opportunities to demonstrate to you how to modify a model by adding extractable objects, adding columns, changing coefficients in an objective function, and changing the type of a variable. When you modify a model by means of the member functions of extractable objects, Concert Technology notifies the

algorithms (instances of subclasses of `IloAlgorithm`, such as `IloCplex` or `IloSolver`) about the modification.

When `IloCplex`, for example, is notified about a change in an extractable object that it has extracted, it maintains as much of the current solution information as it can accurately and reasonably. The *ILOG CPLEX User's Manual* offers more detail about how the algorithm responds to modifications in the model.

Adding Extractable Objects: Both Ways

In *Adding Objects to a Model*, we mention that there are two ways of adding extractable objects to a model: by means of a member function of the model (`IloModel::add`) and by means of a template function (`IloAdd`). In this example, we are motivated to use both ways.

When we add an objective to the model, we need to keep a handle to our objective `RollsUsed` because we need it when we generate columns, so we rely on the template function `IloAdd`, like this:

```
IloObjective   RollsUsed = IloAdd(cutOpt, IloMinimize(env));
```

Apart from the fact that it preserves type information, that single line is equivalent to these lines:

```
IloObjective RollsUsed = IloMinimize(env);
cutOpt.add(RollsUsed);
```

Likewise, we add an array of constraints to our model. We need these constraints later in column generation as well, so we use `IloAdd` again to add the array `Fill` to our model.

```
IloRangeArray  Fill = IloAdd(cutOpt,
                             IloRangeArray(env, amount, IloInfinity));
```

That statement creates `amount.getSize` range constraints. Constraint `Fill[i]` has a lower bound of `amount[i]` and an upper bound of `IloInfinity`.

In contrast, when we create an array of decision variables `Cut`, we use the `add` member function within a loop, like this:

```
IloInt nWdth = size.getSize();
for (j = 0; j < nWdth; j++)
  Cut.add(IloNumVar(RollsUsed(1) + Fill[j](int(rollWidth / size[j]))));
```

In that loop, the variable `Cut[j]` is created with an objective coefficient of 1 (one), and all its pattern coefficients are initially 0 (zero) except for the one in `Fill[j]`. That one is set to `(int(rollWidth / size[j]))`. By default, the type of each variable is `IloFloat`, so we do not explicitly mention it in the loop.

Adding Columns to a Model

Creating a new column to add to a model in Concert Technology is a two-step process:

1. We create a column *expression* defining the new column.
2. We create a *variable* using that column expression and add the variable to the model.

For example, in this problem, `RollsUsed` is an instance of `IloObjective`. The statement `RollsUsed(1)` creates a *term* in a column *expression* defining how to add a new *variable* as a linear term with a coefficient of 1 (one) to the expression `RollsUsed`.

The terms of a column expression are connected to one another by the overloaded `operator +`.

The constraints of this problem are represented in the array `Fill`. A new column to be added to the model has a coefficient for each of the constraints in `Fill`. Those coefficients are represented in the array `newPatt`.

Changing Coefficients of an Objective Function

According to that two-step procedure for adding a column to a model, the following lines create the column with coefficient 1 (one) for the objective `RollsUsed` and create the new variable with bounds at 0 (zero) and at `MAXCUT`.

```
IloNumColumn col = RollsUsed(1);
for (IloInt i = 0; i < Fill.getSize(); ++i)
     col += Fill[i](newPatt[i]);
IloNumVar var(col, 0, MAXCUT);
```

(However, those lines do not appear in the example at hand.) Concert Technology offers a shortcut in the `operator()` for an array of range constraints. Those lines of code can be condensed into the following line:

```
IloNumVar var(RollsUsed(1) + Fill(newPatt), 0, MAXCUT);
```

In other words, `Fill(newPatt)` returns the column expression that the loop would create. You will see a similar shortcut in the example.

There are also other ways of modifying an objective in a Concert Technology model. For example, you can call the member function `IloObjective::setCoef` to change a *coefficient* in an objective. In our example, we use this way of changing a coefficient when we want to find a new pattern.

```
ReducedCost.setCoef(Use, price);
```

You can also change the *sense* of an objective by means of the Concert Technology function `IloMinimize`.

Changing the Type of a Variable

As we explained in *Conversion of a Variable*, in order to change the type of a variable in a model in Concert Technology, you actually create an extractable object (an instance of `IloConversion`, documented in the *ILOG Concert Technology Reference Manual*) and add that object to the model.

In our example, when we want to change the elements of `Cut` (an array of numeric variables) from their default type of `ILOFLOAT` to integer (type `ILOINT`), we create an instance of `IloConversion`, apply it the array `Cut`, and add the conversion to our model, `cutOpt`, like this:

```
cutOpt.add(IloConversion(env, Cut, ILOINT));
```

Cut Optimization Model

Let's summarize our initial model `cutOpt`:

```
IloModel cutOpt (env);

IloObjective   RollsUsed = IloAdd(cutOpt, IloMinimize(env));
IloRangeArray  Fill = IloAdd(cutOpt,
                             IloRangeArray(env, amount, IloInfinity));
IloNumVarArray Cut(env);

IloInt nWdth = size.getSize();
for (j = 0; j < nWdth; j++)
  Cut.add(IloNumVar(RollsUsed(1) + Fill[j](int(rollWidth / size[j]))));
```

Pattern Generator Model

We indicated earlier that we start with an initial model `cutOpt`, and through modifications we eventually build a second model `patGen`. This pattern generator `patGen` (in contrast to `cutOpt`) includes integer variables in the array `Use`. That array appears in the only constraint that we add to `patGen`: a scalar product insuring that the patterns we use do not exceed the width of rolls. We also add a rudimentary objective function to `patGen`. This objective initially consists of only the constant 1 (one). The rest of the objective function depends on the solution we find with the initial model `cutOpt`. We will build that objective function as that information is computed. Here in short is `patGen`:

```
IloModel patGen (env);

IloObjective ReducedCost = IloAdd(patGen, IloMinimize(env, 1));
IloNumVarArray Use(env, nWdth, 0, IloInfinity, ILOINT);
patGen.add(IloScalProd(size, Use) <= rollWidth);
```

Solving the Problem: Using More than One Algorithm

This example does not solve the problem to optimality. It only generates a good feasible solution. It does so by first solving a linear relaxation of the problem. In other words, the decision variables `Cut[j]` are relaxed (not required to be integers). This linear relaxation of the problem then generates columns. As we do so, we keep the variables generated so far, change their type to integer, and resolve the problem.

As you've seen, this example effectively builds two models of the problem, `cutOpt` and `patGen`. Likewise, it uses two algorithms (that is, two instances of `IloCplex`) to arrive at a good feasible solution.

Here's how we create the first algorithm `cutSolver` and extract the initial model `cutOpt`:

```
IloCplex cutSolver(cutOpt);
cutSolver.setRootAlgorithm(IloCplex::Primal);
```

And here is how we create the second algorithm and extract `patGen`:

```
IloCplex patSolver(patGen);
```

The heart of our example is here, in the column generation and optimization over current patterns:

```
IloNumArray price(env, nWdth);
IloNumArray newPatt(env, nWdth);

for (;;) {
  /// OPTIMIZE OVER CURRENT PATTERNS ///

  cutSolver.solve();
  report1 (cutSolver, Cut, Fill);

  /// FIND AND ADD A NEW PATTERN ///

  for (i = 0; i < nWdth; i++)
    price[i] = -cutSolver.getDual(Fill[i]);
  ReducedCost.setCoef(Use, price);

  patSolver.solve();
  report2 (patSolver, Use, ReducedCost);

  if (patSolver.getValue(ReducedCost) > -RC_EPS) break;

  patSolver.getValues(newPatt, Use);
  Cut.add( IloNumVar(RollsUsed(1) + Fill(newPatt)) );
}
cutOpt.add(IloConversion(env, Cut, ILOINT));
cutSolver.solve();
```

In those lines, we solve the current subproblem `cutOpt` by calling `cutSolver.solve`. Then we copy the values of the negative dual solution into the array `price`. We use that

array to set objective coefficients in the model `patGen`. Then we solve the "correct" pattern generation problem.

If the objective value exceeds the negation of the optimality tolerance (`-RC_EPS`), then we have proved that the current solution of the model `cutOpt` is optimal within our optimality tolerance (`RC_EPS`). Otherwise, we copy the solution of the current pattern generation problem into the array `newPatt`, and we use that new pattern to build the next column we add to the model `cutOpt`. Then we repeat our procedure.

Ending the Program

As in all Concert Technology applications, we end this program with a call to `IloEnv::end` to de-allocate the models and algorithms once they are no longer in use.

```
env.end();
```

Complete Program

You can see the entire program here or on-line in the standard distribution of ILOG CPLEX in the file `/examples/src/cutstock.cpp`. To run the example, you need two licenses for ILOG CPLEX. It is also possible to write your own pattern generation models to use with algorithms of ILOG Solver and ILOG Hybrid Optimizers.

```
// ------------------------------------------------------------ -*- C++ -*-
// File: examples/src/cutstock.cpp
// --------------------------------------------------------------------------
//  Copyright (C) 1999-2001 by ILOG.
//  All Rights Reserved.
//  Permission is expressly granted to use this example in the
//  course of developing applications that use ILOG products.
// --------------------------------------------------------------------------
//

#include <ilcplex/ilocplex.h>

ILOSTLBEGIN

#define RC_EPS 1.0e-6

typedef IloArray<IloNumArray> IloNumArray2;


static void readData (const char* filename, IloNum& rollWidth,
                      IloNumArray& size, IloNumArray& amount);
static void report1 (IloCplex& cutSolver,
```

```
                    IloNumVarArray Cut,
                    IloRangeArray Fill);
static void report2 (IloAlgorithm& patSolver,
                    IloNumVarArray Use,
                    IloObjective obj);
static void report3 (IloCplex& cutSolver, IloNumVarArray Cut);



/// MAIN PROGRAM ///

int
main(int argc, char **argv)
{
  IloEnv env;
  try {
    IloInt  i, j;

    IloNum      rollWidth;
    IloNumArray amount(env);
    IloNumArray size(env);

    readData("../../../examples/data/cutstock.dat", rollWidth, size, amount);

    /// CUTTING-OPTIMIZATION PROBLEM ///

    IloModel cutOpt (env);

    IloObjective  RollsUsed = IloAdd(cutOpt, IloMinimize(env));
    IloRangeArray Fill = IloAdd(cutOpt,
                                IloRangeArray(env, amount, IloInfinity));
    IloNumVarArray Cut(env);

    IloInt nWdth = size.getSize();
    for (j = 0; j < nWdth; j++)
      Cut.add(IloNumVar(RollsUsed(1) + Fill[j](int(rollWidth / size[j]))));

    IloCplex cutSolver(cutOpt);
    cutSolver.setRootAlgorithm(IloCplex::Primal);

    /// PATTERN-GENERATION PROBLEM ///

    IloModel patGen (env);

    IloObjective ReducedCost = IloAdd(patGen, IloMinimize(env, 1));
    IloNumVarArray Use(env, nWdth, 0, IloInfinity, ILOINT);
    patGen.add(IloScalProd(size, Use) <= rollWidth);

    IloCplex patSolver(patGen);

    /// COLUMN-GENERATION PROCEDURE ///

    IloNumArray price(env, nWdth);
    IloNumArray newPatt(env, nWdth);
```

```
    /// COLUMN-GENERATION PROCEDURE ///

    for (;;) {
      /// OPTIMIZE OVER CURRENT PATTERNS ///

      cutSolver.solve();
      report1 (cutSolver, Cut, Fill);

      /// FIND AND ADD A NEW PATTERN ///

      for (i = 0; i < nWdth; i++)
        price[i] = -cutSolver.getDual(Fill[i]);
      ReducedCost.setCoef(Use, price);

      patSolver.solve();
      report2 (patSolver, Use, ReducedCost);

      if (patSolver.getValue(ReducedCost) > -RC_EPS) break;

      patSolver.getValues(newPatt, Use);
      Cut.add( IloNumVar(RollsUsed(1) + Fill(newPatt)) );
    }

    cutOpt.add(IloConversion(env, Cut, ILOINT));

    cutSolver.solve();
    report3 (cutSolver, Cut);
    /// END OF PROCEDURE ///
  }
  catch (IloException& ex) {
    cerr << "Error: " << ex << endl;
  }
  catch (...) {
    cerr << "Error" << endl;
  }

  env.end();

  return 0;
}


static void readData (const char* filename, IloNum& rollWidth,
                      IloNumArray& size, IloNumArray& amount)
{
   ifstream in(filename);
   if (in) {
      in >> rollWidth;
      in >> size;
      in >> amount;
   }
   else {
```

```
      cerr << "No such file: " << filename << endl;
      throw(1);
   }
}

static void report1 (IloCplex& cutSolver,
                     IloNumVarArray Cut,
                     IloRangeArray Fill)
{
   cout << endl;
   cout << "Using " << cutSolver.getObjValue() << " rolls" << endl;
   cout << endl;
   for (IloInt j = 0; j < Cut.getSize(); j++)
      cout << "  Cut" << j << " = " << cutSolver.getValue(Cut[j]) << endl;
   cout << endl;
   for (IloInt i = 0; i < Fill.getSize(); i++)
      cout << "  Fill" << i << " = " << cutSolver.getDual(Fill[i]) << endl;
   cout << endl;
}

static void report2 (IloAlgorithm& patSolver,
                     IloNumVarArray Use,
                     IloObjective obj)
{
   cout << endl;
   cout << "Reduced cost is " << patSolver.getValue(obj) << endl;
   cout << endl;
   if (patSolver.getValue(obj) <= -RC_EPS) {
      for (IloInt i = 0; i < Use.getSize(); i++)
         cout << "  Use" << i << " = " << patSolver.getValue(Use[i]) << endl;
      cout << endl;
   }
}

static void report3 (IloCplex& cutSolver, IloNumVarArray Cut)
{
   cout << endl;
   cout << "Best integer solution uses "
        << cutSolver.getObjValue() << " rolls" << endl;
   cout << endl;
   for (IloInt j = 0; j < Cut.getSize(); j++)
      cout << "  Cut" << j << " = " << cutSolver.getValue(Cut[j]) << endl;
}
```

C H A P T E R

6

Rates: Using ILOG CPLEX and Semi-Continuous Variables

This chapter uses an example of managing production in a power plant to demonstrate semi-continuous variables in Concert Technology. In it, you will learn:

- more about `IloCplex`, the C++ class giving you access to ILOG CPLEX;
- more about building a model with Concert Technology classes;
- how to use the class `IloSemiContVar`.

What Is IloCplex?

The class `IloCplex`, documented in the *ILOG CPLEX Reference Manual*, derives from the class `IloAlgorithm`, documented in the *ILOG Concert Technology Reference Manual*. An instance of `IloCplex` is capable of solving optimization problems known as mixed integer programs (MIPs) and linear programs (LPs).

IloCplex and Extraction

As a Concert Technology algorithm, an instance of `IloCplex` extracts the following classes from a model when you call its member function `IloCplex::extract`:

- `IloNumVar` representing numeric variables;
- `IloSemiContVar` representing semi-continuous or semi-integer variables;
- `IloObjective` representing at most one objective function in linear or piecewise linear expressions;
- `IloRange` representing range constraints in linear or piecewise linear expressions;
- `IloConversion` representing type conversions of variables (from floating-point to integer, for example);
- `IloSOS1` representing special ordered sets of type 1;
- `IloSOS2` representing special ordered sets of type 2;
- `IloAnd` for use with `IloSolution`.

IloCplex and MP Models

What is special about that list of Concert Technology classes recognized by an instance of `IloCplex`? A model consisting of instances of those classes can be transformed into a MIP or LP in the conventional form:

Maximize (or minimize) an objective function

such that `LowerBounds <= Ax <= UpperBounds`

for every `lowerBound <= x <= upperBound`

When all the variables, indicated by `x`, are continuous floating-point variables, a problem in this conventional form is known as a linear programming model (LP). When some variables are integer- or Boolean-valued, a problem in this form is known as a mixed integer programming model (MIP).

At first glance, it might appear that this formulation greatly restricts the kinds of problems that can be modeled in this way. However, practice has shown that an astonishing variety of problems can be represented by such a model. The book *Model Building in Mathematical Programming* by H.P. Williams offers a good starting point if you are interested in model-building techniques for LPs and MIPs.

IloCplex and Algorithms

An important observation to make here is that an instance of `IloCplex` is not really *one* algorithm, though we refer to it as an algorithm since `IloCplex` is a subclass of `IloAlgorithm`. In fact, an instance of `IloCplex` consists of a set of highly configurable algorithms (also known as optimizer options). They include primal and dual simplex algorithms, barrier algorithms, a network simplex solver and a branch & cut solver for MIPs. In most cases, you can use `IloCplex` like a “black box” without concern about the options, though you can also select the optimizer options individually yourself. The options provide a

wealth of parameters that allow you to fine-tune the algorithm to your particular model. The *ILOG CPLEX User's Manual* contains detailed instructions about exploiting these options.

In the case of MIPs, for example, you can use your own callback directly to control the branch & cut search carried out by `IloCplex`.

In pure LPs (that is, linear programs with no integer variables, no Boolean variables, no semi-continuous variables, no piecewise linear functions, no special ordered sets (SOSs)), you can query `IloCplex` for additional information, such as dual information or basis information (with appropriate options). `IloCplex` also supports sensitivity analysis indicating how to modify your model while preserving the same solution. In cases where your "solution" is infeasible, the infeasibility finder enables you to analyze the source of infeasibility.

What Are Semi-Continuous Variables?

A semi-continuous variable is a variable that by default can take the value 0 (zero) or any value between its semi-continuous lower bound (sclb) and its upper bound (ub). Both those bounds—the sclb and the ub—must be finite. In Concert Technology, semi-continuous variables are represented by the class `IloSemiContVar`. To create a semi-continuous variable, you use the constructor from that class to indicate the environment, the semi-continuous lower bound, and the upper bound of the variable, like this:

```
IloSemiContVar mySCV(env, 1.0, 3.0);
```

That line creates a semi-continuous variable with a semi-continuous lower bound of 1.0 and an upper bound of 3.0. The member function `IloSemiContVar::getSemiContinuousLb` returns the semi-continuous lower bound of the invoking variable, and the member function `IloSemiContVar::getUb` returns the upper bound. That class, its constructors, and its member functions are documented in the *ILOG Concert Technology Reference Manual*.

In that manual, you will see that `IloSemiContVar` derives from `IloNumVar`, the Concert Technology class for numeric variables. Like other numeric variables, semi-continuous variables assume floating-point values by default (type `ILOFLOAT`). However, you can designate a semi-continuous variable as integer (type `ILOINT`). In that case, we refer to it as a semi-integer variable.

For details about the feasible region of a semi-continuous or semi-integer variable, see the documentation of `IloSemiContVar` in the *ILOG Concert Technology Reference Manual*.

Describing the Problem

With this background about semi-continuous variables, let's look at an example using them. We will assume that we are managing a power plant of several generators. Each of the generators may be on or off (producing or not producing power). When a generator is on, it produces power between its minimum and maximum level, and each generator has its own minimum and maximum levels. The cost for producing a unit of output differs for each generator as well. The aim of our problem is to best satisfy demand for power while minimizing cost.

Representing the Problem

As input for this example, we need such data as the minimum and maximum output level for each generator. We will use Concert Technology arrays `minArray` and `maxArray` for that data. We will read data from a file into these arrays, and then learn their length (that is, the number of generators available to us) by calling the member function `getSize`.

We also need to know the cost per unit of output for each generator. Again, a Concert Technology array, `cost`, serves that purpose as we read data in from a file with the `operator >>`.

We also need to know the demand for power, which we will represent as a numerical variable, `demand`.

Building a Model

Once we have created an environment and a model in that environment, we are ready to populate the model with extractable objects pertinent to our problem.

We will represent the production level of each generator as a semi-continuous variable. In that way, with the value 0 (zero), we can accommodate whether the generator is on or off; with the semi-continuous lower bound of each variable, we can indicate the minimum level of output from each generator; and we indicate the maximum level of output for each generator by the upper bound of its semi-continuous variable. The following lines create the array `production` of semi-continuous variables (one for each generator), like this:

```
IloNumVarArray production(env);
   for (IloInt j = 0; j < generators; ++j)
        production.add(IloSemiContVar(env, minArray[j], maxArray[j]));
```

We add an objective to our model to minimize production costs in this way:

```
mdl.add(IloMinimize(env, IloScalProd(cost, production)));
```

We also add a constraint to the model: we must meet demand.

```
mdl.add(IloSum(production) >= demand);
```

With that model, now we are ready to create an algorithm (in this case, an instance of `IloCplex`) and extract the model.

Solving the Problem

To solve the problem, we simply follow the steps outlined in *Solving the Problem*: we create the algorithm, extract the model, and solve.

Ending the Program

As in all Concert Technology applications, we end this program with a call to `IloEnv::end` to de-allocate the model and algorithm once they are no longer in use.

```
env.end();
```

Complete Program

You can see the entire program here or on-line in the standard distribution of ILOG CPLEX in the file `/examples/src/rates.cpp`. To run that example, you need a license for ILOG CPLEX.

```
// ------------------------------------------------------------ -*- C++ -*-
// File: examples/src/rates.cpp
// --------------------------------------------------------------------------
//  Copyright (C) 1999-2001 by ILOG.
//  All Rights Reserved.
//  Permission is expressly granted to use this example in the
//  course of developing applications that use ILOG products.
// --------------------------------------------------------------------------

#include <ilcplex/ilocplex.h>
ILOSTLBEGIN

int
main(int argc, char** argv)
{
  IloEnv env;
  try {
    IloNumArray minArray(env), maxArray(env), cost(env);
    IloNum demand;
```

```
    if ( argc < 2 ) {
      env.warning()
        << "Default data file : ../../../examples/data/rates.dat" << endl;
      ifstream in("../../../examples/data/rates.dat");
      in >> minArray >> maxArray >> cost >> demand;
    } else {
      ifstream in(argv[1]);
      in >> minArray >> maxArray >> cost >> demand;
    }
    IloInt generators = minArray.getSize();

    IloModel mdl(env);

    IloNumVarArray production(env);
    for (IloInt j = 0; j < generators; ++j)
      production.add(IloSemiContVar(env, minArray[j], maxArray[j]));

    mdl.add(IloMinimize(env, IloScalProd(cost, production)));
    mdl.add(IloSum(production) >= demand);

    IloCplex cplex(mdl);
    cplex.exportModel("rates.lp");
    if (cplex.solve()) {
      for (IloInt j = 0; j < generators; ++j) {
         cplex.out() << "   generator " << j << ": "
                     << cplex.getValue(production[j]) << endl;
      }
      cplex.out() << "Total cost = " << cplex.getObjValue() << endl;
    }
    else cplex.out()<< "No solution" << endl;
    cplex.printTime();
  }
  catch (IloException& ex) {
    cerr << "Error: " << ex << endl;
  }
  catch (...) {
    cerr << "Error" << endl;
  }
  env.end();
  return 0;
}
```

C H A P T E R

7

Car Sequencing: Using ILOG Solver

This chapter presents a car sequencing problem to demonstrate constraint programming facilities in ILOG Solver and Concert Technology. In it, you will see:

- how to express capacity constraints in a car sequencing problem;
- how to use predefined ILOG Solver constraints such as `IloDistribute` and `IloAbstraction`.

Describing the Problem

Originally, car sequencing problems arose on assembly lines in the automotive industry, where it is possible to build many different types of cars, each type corresponding to a basic model with various added options. In other words, we may think of one type of vehicle as a particular *configuration* of options. While the car is on the assembly line, it is possible to put many options on the same vehicle.

In theory, that is, any possible configuration could be produced on an assembly line, but for practical reasons (such as cost, time, or customer preference) a given option will not be installed on every vehicle. We call this constraint—not every option on every vehicle—the *capacity* of an option. It is usually represented as a ratio *p/q*, where *q* is any sequence of cars on the line, and *p* is the greatest number of cars with that option.

The problem in car sequencing then consists of our determining in which *order* cars corresponding to each configuration will be assembled, while we keep in mind that we must build a certain number of cars per configuration.

Representing the Data

To make our explanation clear, we will keep this example small and simple; for a more robust version that scales up to industrial standards, see the car-sequencing example in the *ILOG Solver User's Manual*.

For this example, we assume that there are ten cars to build, five options available for installation, and six required configurations. Table 7.1 indicates which options belong to which configuration: • means that configuration *j* requires option *i*; a blank means configuration *j* does not require option *i*. The table also shows the capacity of each option as well as the number of cars to build for each configuration.

Table 7.1 *Data for Car Sequencing*

Option	Capacity	Configurations					
		0	1	2	3	4	5
0	1/2	•				•	•
1	2/3			•	•		•
2	1/3	•				•	
3	2/5	•	•		•		
4	1/5			•			
number of cars		1	1	2	2	2	2

For example, the table indicates that option 1 can be put on at most two cars for any sequence of three cars. Option 1 is required by configurations 2, 3, and 5.

We'll create a constrained variable for each car that should be assembled on the line. Consequently, we'll have ten constrained variables that we'll organize into an array, `cars`. We'll assume that the order in which the cars appear on the line is the same as the order of the constrained variables in the array. That is, `cars[0]` will be the first car; `cars[9]` will be the last.

The values that the constrained variables can assume correspond to the configurations. If a constrained variable is bound to the value 3, for example, then the corresponding car will have options corresponding to configuration 3. Initially, a car can have any configuration.

Thus we have the following code so far:

```
const IloInt nbOptions = 5;
const IloInt nbConfs   = 6;
const IloInt nbCars    = 10;

IloIntVarArray cars(env, nbCars,0,nbConfs-1);

IloIntArray confs(env, 6, 0, 1, 2, 3, 4, 5);
IloIntArray nbRequired(env, 6, 1, 1, 2, 2, 2, 2);
```

Developing a Model

As a first step in developing a model for this problem, we create a Concert Technology environment, an instance of `IloEnv`, and a model, an instance of `IloModel`.

```
IloEnv env;
IloModel model(env);
```

In that environment, we will create the constraints of the problem and add them to the model.

Introducing IloDistribute

The number of cars required for each configuration can be expressed by means of the predefined constraint `IloDistribute`. This function takes three parameters: an array of constrained variables, an array of constrained values, and an array to count those constrained values (the cardinalities). The following code actually implements the specific details of our example.

```
IloIntVarArray cards(env, 6);
for(IloInt conf=0;conf<nbConfs;conf++) {
  cards[conf]=IloIntVar(env, nbRequired[conf],nbRequired[conf]);
}

model.add (IloDistribute(env, cards, confs, cars));
```

In this case, the constrained variables in the array `cards` are equal to the number of occurrences in the array `cars` of the values in the array `confs` (our configurations). More precisely, for each `i`, `cards[i]` is equal to the number of occurrences of `confs[i]` in the array `cars`. In this way, we can manage the number of times (`cards[i]`) that a given configuration of options (`conf[i]`) is produced as a car (an element of the array `cars`) on the assembly line. After propagation of this constraint, the minimum of `cards[i]` is at least equal to the number of variables contained in `cars` bound to the value at `confs[i]`; and the

maximum of `cards[i]` is at most equal to the number of variables contained in `cars` that contain the value at `confs[i]` in their domain.

Introducing IloAbstraction

To see how the predefined constraint `IloAbstraction` works, let's assume that `x` is an array of two constrained variables and that the domains of those two variables are `{0, 1, 2, 4, 6}` and `{0, 2, 3, 5, 6}`. Those two variables and their domains will be passed as the argument `x` to `IloAbstraction`. Let's also suppose that we're interested only in the values 0, 1, 2, and 3 in those two domains. Those will be the `values` array passed as an argument to the constructor of `IloAbstraction`. We will arbitrarily choose 7 as the abstract value to pass to the constructor of `IloAbstraction`. This abstract value (7) will appear like a joker in places where the other values that interest us { 0, 1, 2, 3} are not available. Then, to get the result we want, we'll use the constraint `IloAbstraction` like this, to fill in the array `y`:

```
IloAbstraction(env, y, x, values, 7);
```

In this example, 0 occurs in the domain of `x[0]` and in `values`, so it will appear in the domain of `y[0]` as well. Likewise, 1 and 2 occur in the domain of `x[0]` and in `values`, so they, too, appear in the domain of `y[0]`. The numbers 4 and 6 occur in the domain of `x[0]`; however, they do not appear in the array `values`, so they do not appear in the domain of `y[0]`; instead, for those numbers, the abstract value 7 appears in the domain of `y[0]`. (It occurs only once because of the nature of a domain; a domain resembles a set more than an array in this respect.)

`IloAbstraction` builds the domain of `y[1]` in a similar way. The numbers 0, 2, and 3 appear both in the domain of `x[1]` and in the array `values`, so they become part of the domain of `y[1]`. The numbers 5 and 6 from the domain of `x[1]` are represented in the domain of `y[1]` by the abstract value 7 because they do not figure in the array `values`.

In short, the two constrained variables in the array `y` will have the domains `{0, 1, 2, 7}` and `{0, 2, 3, 7}`.

In *Using IloAbstraction*, you will see how we exploit this predefined constraint in our car sequencing application.

Notation

Let's summarize our notation so far:

- `optConf[i]` is an array containing the configurations that require option i. We use the template `IloArray` predefined in Concert Technology to declare `optConf`.

```
IloArray<IloIntArray> optConf(env, nbOptions);

optConf[0] = IloIntArray(env, 3, 0, 4, 5);
optConf[1] = IloIntArray(env, 3, 2, 3, 5);
optConf[2] = IloIntArray(env, 2, 0, 4);
optConf[3] = IloIntArray(env, 3, 0, 1, 3);
optConf[4] = IloIntArray(env, 1, 2);
```

- `maxSeq` and `overSeq` are two arrays; together, they express the capacity constraint on options; `maxSeq[i]/overSeq[i]` represents the capacity of option i.

```
IloIntArray maxSeq(env, 5, 1, 2, 1, 2, 1);
IloIntArray overSeq(env, 5, 2, 3, 3, 5, 5);
```

- `nbRequired` is an array defining the number of cars that have to be built for each configuration. We initialize its values with data from the problem.

```
IloIntArray nbRequired(env, 6, 1, 1, 2, 2, 2, 2);
```

- `optCard[i]` is an array containing the cardinality variables associated with option i. Again, we use the template `IloArray` predefined in Concert Technology to declare `optCard`, and we initialize its values with problem data.

```
IloArray<IloIntArray> optCard(env, nbOptions);
optCard[0] = IloIntArray(env, 3, 1, 2, 2);
optCard[1] = IloIntArray(env, 3, 2, 2, 2);
optCard[2] = IloIntArray(env, 2, 1, 2);
optCard[3] = IloIntArray(env, 3, 1, 1, 2);
optCard[4] = IloIntArray(env, 1, 2);
```

With that notation, we can write the following code.

```
for (IloInt opt=0; opt < nbOptions; opt++) {
  for (IloInt i=0; i < nbCars-overSeq[opt]+1; i++) {
    IloIntVarArray sequence(env,overSeq[opt]);
    for (IloInt j=0; j < overSeq[opt]; j++)
      sequence[j] = cars[i+j];
    model.add(IloCarSequencing(env,
                               (IloInt)maxSeq[opt],
                               optCard[opt],
                               optConf[opt],
                               sequence));
  }
}
```

In the following section, you'll see how we define our own `IloCarSequencing` by using the predefined constraints `IloDistribute` and `IloAbstraction` as building blocks.

Defining Your Own IloCarSequencing

So how does a user define `IloCarSequencing`? Within an environment (an instance of `IloEnv`), we exploit predefined constraints, `IloAbstraction` and `IloDistribute`, (documented in the *ILOG Concert Technology Reference Manual*), as building blocks in the constraint we need for our particular problem. The following lines offer you an overview of `IloCarSequencing`, and the sections after this code walk you through the details.

```
IloModel IloCarSequencing( IloEnv env,
                           IloInt maxCar,
                           IloIntArray maxConfs,
                           IloIntArray confs,
                           IloIntVarArray sequence){

  const IloInt abstractValue = -1;
  const IloInt confs_size  =confs.getSize();
  const IloInt sequence_size = sequence.getSize();
  IloIntArray nval(env, confs_size+1);
  IloIntVarArray ncard(env, confs_size+1);

  nval[OL]=abstractValue;
  ncard[OL]=IloIntVar(env, sequence_size - maxCar, sequence_size);

  IloInt i;
  for(i=0;i<confs_size;i++){
    nval[i+1]=confs[i];
    if (maxConfs[i] > maxCar)
      ncard[i+1]=IloIntVar(env, 0,maxCar);
    else
      ncard[i+1]=IloIntVar(env, 0,maxConfs[i]);
  }

  IloIntVarArray nvars(env,sequence_size);
  for (i = 0; i < sequence_size ; i++)
    nvars[i] = IloIntVar(env, nval);

  IloModel carseq_model(env);
  carseq_model.add(IloAbstraction(env, nvars, sequence, confs,abstractValue));
  carseq_model.add(IloDistribute(env, ncard, nval, nvars));
  return carseq_model;
}
```

Using IloAbstraction

Starting from the array `maxConfs`, we can build an array of constrained integer variables `nCard` so that the minimum value in the domain of each variable is 0, and the maximum value in the domain of the variable `nCard[i]` is `maxConfs[i]`.

Let's call the number of variables contained in `carsseq` "`nbCarsseq`." If `maxCar` is equal to `nbCarsseq`, then the constraint can be written quite simply as `IloDistribute(nCard,confs,carsseq);`. However, if `maxCar` is less than

nbCarsseq, then the constraint imposes the condition that the values that are not in the array confs must be taken at least (nbCarsseq - maxCar) times.

Reasoning accordingly, we'll formulate a cardinality constraint on the set of values that are not in confs. For that purpose, we'll use the Concert Technology function IloAbstraction. This function abstracts a set of values by means of a given value, known as the abstraction value.

The abstraction value thus represents all the values that are *not* in the array of given values. With such a handy value, we will be able to constrain the number of times that the value is taken; we'll do so by means of IloDistribute.

Using IloDistribute

As the abstraction value for our car-sequencing problem, we'll choose the maximum of all the domains plus one. The array of variables that we will use in the function IloDistribute will thus be defined like this:

```
IloIntArray nval(env, confs_size+1);
IloIntVarArray ncard(env, confs_size+1);
```

Next it suffices for us to create an array of values corresponding to the array of configurations and to put our abstraction value into it. Then, we need to create the variables expressing the number of times that these values can be taken. If maxCar is greater than the number of times that a given configuration should be built, then we can immediately constrain it by that second number.

```
IloInt i;
for(i=0;i<confs_size;i++){
  nval[i+1]=confs[i];
  if (maxConfs[i] > maxCar)
    ncard[i+1]=IloIntVar(env, 0,maxCar);
  else
    ncard[i+1]=IloIntVar(env, 0,maxConfs[i]);
}
```

In our user-defined IloCarSequencing, we create a model and add these specialized constraints to it, like this:

```
IloModel carseq_model(env);
carseq_model.add(IloAbstraction(env, nvars, sequence, confs,abstractValue));
carseq_model.add(IloDistribute(env, ncard, nval, nvars));
return carseq_model;
```

The practical effect of this combination is that we achieve greater pruning of our search tree by exploiting IloCarSequencing.

Solving the Problem

With the predefined constraints and our own `IloCarSequencing`, all added to the model, we solve the problem in a few lines of Concert Technology code. We simply create an algorithm (an instance of `IloSolver`, for example) in the environment, extract the model for that algorithm, and solve with a predefined goal `IloGenerate`, like this:

```
IloSolver solver(model);

solver.solve(IloGenerate(env, cars, IlcChooseMinSizeInt));
```

The predefined goal `IloGenerate` and the predefined search criterion `IloChooseMinSizeNum` are documented in the *ILOG Solver Reference Manual*. This search criterion chooses a variable with the smallest domain from among those available. The goal generates a search tree based on the array of variables `cars` and the search criterion.

Displaying a Solution

To display the solution, we use the output operator of an instance of `IloSolver` to display the values of constrained variables that interest us. We also use the predefined Solver member function `IloSolver::printInformation` to display details about the search.

```
solver.out() << “cars = “ << solver.getIntVarArray(cars) << endl;

solver.printInformation();
```

Ending the Program

To indicate to Concert Technology that our application has completed its use of the environment, we call the member function `IloEnv::end`. This member function cleans up the environment, de-allocating memory used in the model and the search for a solution.

```
env.end();
```

Complete Program

You can see the complete program here or on line in the standard distribution of ILOG Solver in the file `/examples/src/carseq-simple.cpp`. To run that example, you need a license for ILOG Solver. There is also a more complicated "industrial strength" version explained in the *ILOG Solver 5.1 User's Manual* and available as an example on line in the file `/examples/src/carseq-simple.cpp`.

```
#include <ilsolver/ilosolver.h>

ILOSTLBEGIN

IloModel IloCarSequencing( IloEnv env,
                           IloInt maxCar,
                           IloIntArray maxConfs,
                           IloIntArray confs,
                           IloIntVarArray sequence){

  const IloInt abstractValue = -1;
  const IloInt confs_size  =confs.getSize();
  const IloInt sequence_size = sequence.getSize();
  IloIntArray nval(env, confs_size+1);
  IloIntVarArray ncard(env, confs_size+1);

  nval[0L]=abstractValue;
  ncard[0L]=IloIntVar(env, sequence_size - maxCar, sequence_size);

  IloInt i;
  for(i=0;i<confs_size;i++){
    nval[i+1]=confs[i];
    if (maxConfs[i] > maxCar)
      ncard[i+1]=IloIntVar(env, 0,maxCar);
    else
      ncard[i+1]=IloIntVar(env, 0,maxConfs[i]);
  }

  IloIntVarArray nvars(env,sequence_size);
  for (i = 0; i < sequence_size ; i++)
    nvars[i] = IloIntVar(env, nval);

  IloModel carseq_model(env);
  carseq_model.add(IloAbstraction(env, nvars, sequence, confs,abstractValue));
  carseq_model.add(IloDistribute(env, ncard, nval, nvars));
  return carseq_model;
}

int main() {
  try {
    IloEnv env;
    IloModel model(env);
```

```
const IloInt nbOptions = 5;
const IloInt nbConfs   = 6;
const IloInt nbCars    = 10;

IloIntVarArray cars(env, nbCars,0,nbConfs-1);

IloIntArray confs(env, 6, 0, 1, 2, 3, 4, 5);
IloIntArray nbRequired(env, 6, 1, 1, 2, 2, 2, 2);

IloIntVarArray cards(env, 6);
for(IloInt conf=0;conf<nbConfs;conf++) {
  cards[conf]=IloIntVar(env, nbRequired[conf],nbRequired[conf]);
}

model.add (IloDistribute(env, cards, confs, cars));

IloArray<IloIntArray> optConf(env, nbOptions);

optConf[0] = IloIntArray(env, 3, 0, 4, 5);
optConf[1] = IloIntArray(env, 3, 2, 3, 5);
optConf[2] = IloIntArray(env, 2, 0, 4);
optConf[3] = IloIntArray(env, 3, 0, 1, 3);
optConf[4] = IloIntArray(env, 1, 2);

IloIntArray maxSeq(env, 5, 1, 2, 1, 2, 1);
IloIntArray overSeq(env, 5, 2, 3, 3, 5, 5);

IloArray<IloIntArray> optCard(env, nbOptions);
optCard[0] = IloIntArray(env, 3, 1, 2, 2);
optCard[1] = IloIntArray(env, 3, 2, 2, 2);
optCard[2] = IloIntArray(env, 2, 1, 2);
optCard[3] = IloIntArray(env, 3, 1, 1, 2);
optCard[4] = IloIntArray(env, 1, 2);

for (IloInt opt=0; opt < nbOptions; opt++) {
  for (IloInt i=0; i < nbCars-overSeq[opt]+1; i++) {
    IloIntVarArray sequence(env,overSeq[opt]);
    for (IloInt j=0; j < overSeq[opt]; j++)
      sequence[j] = cars[i+j];
    model.add(IloCarSequencing(env,
                               (IloInt)maxSeq[opt],
                               optCard[opt],
                               optConf[opt],
                               sequence));
  }
}

IloSolver solver(model);

solver.solve(IloGenerate(env, cars, IlcChooseMinSizeInt));

solver.out() << “cars = “ << solver.getIntVarArray(cars) << endl;
```

```
    solver.printInformation();

    env.end();
  }
  catch (IloException& ex) {
    cerr << "Error: " << ex << endl;
  }
  return 0;
}
```


Results

Here are typical results of running that simple version of a car-sequencing application:

```
cars = IlcIntVarArrayI[[0] [1] [5] [2] [4] [3] [3] [4] [2] [5]]
Number of fails               : 1
Number of choice points       : 3
Number of variables           : 308
Number of constraints         : 239
Reversible stack (bytes)      : 28164
Solver heap (bytes)           : 172884
Solver global heap (bytes)    : 4044
And stack (bytes)             : 4044
Or stack (bytes)              : 4044
Search Stack (bytes)          : 4044
Constraint queue (bytes)      : 11144
Total memory used (bytes)     : 228368
Running time since creation   : 0.03
```

The details about failures, choice points, memory use, running time, and so forth are readily available from the member function `IloSolver::printInformation`, documented in the *ILOG Solver Reference Manual*.

C H A P T E R

8

Dispatching Technicians: Using ILOG Dispatcher

This chapter presents a problem of dispatching technicians to demonstrate ILOG Dispatcher facilities with Concert Technology. In it, you will learn:

- how to use intrinsic dimensions to match technician skills to customer needs;
- how to use local search to improve a solution.

The example in this chapter also appears in the *ILOG Dispatcher User's Manual*, where it is a first step in an exposition of more complicated dispatching solutions. If this introductory example interests you, you might also want to consider the extended versions available in the *ILOG Dispatcher User's Manual*.

Describing the Problem

ILOG Dispatcher is a C++ library of classes and functions adapted to vehicle routing and dispatching problems. The Dispatcher library is designed to work in conjunction with Concert Technology to represent aspects of dispatching and routing problems: routing plans, visits, vehicles, and their constraints, such as capacity or time-window constraints. For more detail about dispatching and routing problems and their solutions, see the *ILOG Dispatcher Reference* and *User's Manuals*.

One variety of dispatching problem concerns people who service, repair, or install equipment at various sites. In purely impersonal terms, these technicians resemble vehicles in a vehicle routing problem (VRP). As a vehicle may deliver or pick up quantities of goods, it may also be accompanied by a technician appropriate to the customer's needs. However, technicians generally exercise special skills or sets of skills required by a customer, and in that context, an effective model of the problem must also represent those skills and apply constraints and costs to them.

The example in this chapter treats technicians with multiple skills. Technicians with the right skill must be dispatched to jobs that require that skill. The example is based on a standard VRP with the additional consideration that each vehicle contains a technician with a specific skill, and each visit requires a specific level of skill.

In this example, each technician can exercise only one skill at a time. Therefore, one visit must be made for each required skill. For a visit type with skill requirements 4, 0, 0, one visit will be made; for a visit type with skill requirements 2, 0, 2, two visits will be made, and so forth.

Representing the Data

In some problems, it is convenient to read data from a file, as you've seen in other examples in this manual. In other problems, the application includes its data. This example uses both strategies.

Using Data in the Application

For this example, we assume that there are three skills required; ten different types of visits; and five types of technicians. We represent this data within the application in two matrices (that is, in two-dimensional arrays), like this:

```
const IloInt NbOfSkills = 3;
const IloInt NbOfVisitTypes = 10;
const IloInt NbOfTechTypes = 5;


const IloInt Skills[NbOfTechTypes][NbOfSkills] = {{ 0, 2, 5},
                                                  { 3, 0, 4},
                                                  { 2, 3, 2},
                                                  { 4, 2, 1},
                                                  { 4, 4, 0}};

const IloInt RequiredSkills[NbOfVisitTypes][NbOfSkills] = {{ 4, 0, 0},
                                                           { 2, 1, 0},
                                                           { 4, 2, 5},
                                                           { 3, 0, 0},
                                                           { 1, 1, 0},
                                                           { 0, 4, 1},
                                                           { 1, 2, 3},
                                                           { 3, 3, 0},
                                                           { 0, 0, 2},
                                                           { 2, 0, 2}};
```

The matrix `Skills` represents the technicians and the level of their skills. A value of 0 (zero) in this matrix indicates that the technician lacks this skill. The matrix `RequiredSkills` represents the types of visits by levels of skill required. In this matrix, a value of 0 (zero) indicates that skill is not required for that visit.

Reading Data from a File

As in earlier examples in this manual, we will read part of the data for this problem from a file. We use the stream `infile` to enter data indicating the number of trucks and their capacity as well as the number of visits. We also read coordinates of depots from a file so

that we can compute the distance between depots and other locations. The file also contains data about the opening and closing times of depots.

```
ifstream infile;
char * problem;
if (argc >=2) problem = argv[1];
else          problem = (char *) "../../../examples/data/vrp20.dat";
infile.open(problem);
if (!infile) {
  env.out() << "File not found or not specified: " << problem << endl;
  env.end();
  return 0;
}
IloInt nbOfVisits, nbOfTrucks;
infile >> nbOfVisits >> nbOfTrucks;
IloNum capacity;
infile >> capacity;
IloNum depotX, depotY, openTime, closeTime;
infile >> depotX  >> depotY >> openTime >> closeTime;
```

Developing a Model

As in other Concert Technology applications, we begin by creating an environment and within the environment, we also create a model, like this:

```
IloEnv env;
try {
  IloModel mdl(env);
```

Within this environment, we create a depot, that is, a node representing a geographic location, where all vehicles (carrying technicians to customer sites) start and end their routes.

```
IloNode depot(env, depotX, depotY);
```

Using Intrinsic Dimensions to Represent Levels of Skills

In ILOG Dispatcher, *intrinsic dimensions* represent quantities that do not depend on outside factors. In typical VRPs, intrinsic dimensions represent weight, volume, or quantity of goods to pick up or to deliver. However, in this problem, we use several intrinsic dimensions in our

environment to represent different skills—both those possessed by technicians and those required at visits.

```
IloDimension2 time(env, IloEuclidean, “Time”);
mdl.add(time);
IloDimension1 skill1(env, “skill1”);
mdl.add(skill1);
IloDimension1 skill2(env, “skill2”);
mdl.add(skill2);
IloDimension1 skill3(env, “skill3”);
mdl.add(skill3);
IloDimension1 skills[NbOfSkills] = {skill1, skill2, skill3};
```

The extrinsic dimension `Time` uses Euclidean distance. ILOG Dispatcher offers predefined classes and functions for calculating and manipulating a variety of measures, such as Euclidean distance.

Adding Constraints to the Model

Now we create a starting visit (first) and ending visit (last) at the depot for each technician. We stipulate that the visit starting from the depot cannot begin before the depot opens (`openTime`). Likewise, the ending visit must reach the depot again before it closes (`closeTime`). We add those stipulations to our model as constraints.

```
for (IloInt j = 0; j < nbOfTrucks * NbOfSkills; j++) {
  IloVisit first(depot, “Depot”);
  mdl.add(first.getCumulVar(time) >= openTime);
  IloVisit last(depot, “Depot”);
  mdl.add(last.getCumulVar(time) <= closeTime);
```

In this problem, the cost of a technician is directly proportional to the time spent on the visit (`time`).

```
char name[16];
sprintf(name, “Vehicle %d”, j);
IloVehicle technician(first, last, name);
technician.setCost(time, 1.0);
```

We also add a constraint to set the technician’s level of skill at the first visit. This level does not change at each visit, because the vehicle/technician does not gain or lose skill at the visits (in contrast, say, to the weight of goods picked up or delivered in a conventional VRP).

```
for (IloInt k = 0; k < NbOfSkills; k++) {
  mdl.add(first.getCumulVar(skills[k]) == Skills[j % NbOfTechTypes][k]);
}
mdl.add(technician);
```

The modulo operator (%) converts a vehicle index to the technician type. Since there are five types of technicians, this conversion assigns vehicle 0 to type 0, vehicle 1 to type 1, vehicle 5 to type 0, vehicle 6 to type 1, and so on.

We use the modulo operator (%) again to convert the visit index to the visit type. There are ten types of visits; visit 0 is type 0, visit 1 is type 1, visit 10 is type 0, visit 11 is type 1, and so forth. We create a node for the visit. Then we create the visits. Each visit takes a certain amount of time (`dropTime`), and it must be performed within a given time window (between its `minTime` and `maxTime`). We also stipulate that the technician making the visit must have at least the required level of skill. We add those constraints about visits to the model.

```
for (IloInt i = 0; i < nbOfVisits; i++) {
  IloInt id; // visit identifier
  IloNum x, y, quantity, minTime, maxTime, dropTime;
  infile >> id >> x >> y >> quantity >> minTime >> maxTime >> dropTime;
  for (IloInt k = 0; k < NbOfSkills; k++) {
    IloInt requiredSkill = RequiredSkills[i % NbOfVisitTypes][k];
    if (requiredSkill > 0) {
      IloNode customer(env, x, y);
      char name[16];
      sprintf(name, "%d(%d)", id, k+1);
      IloVisit visit(customer, name);
      mdl.add(visit.getDelayVar(time) == dropTime);
      mdl.add(minTime <= visit.getCumulVar(time) <= maxTime);
      mdl.add(visit.getCumulVar(skills[k]) >= requiredSkill);
      mdl.add(visit);
    }
  }
}
```

Solving the Problem: Using Local Search to Improve a Solution

In this example, we generate an initial solution and then improve it. To do so, we use the local search facilities available in ILOG Solver and Dispatcher. For more detail about these local search facilities, see the *ILOG Solver Reference Manual* as well as the *ILOG Dispatcher Reference* and *User's Manuals*.

First we use the predefined goal `IloInsertionGenerate`; we add it to our model, solve for a preliminary solution, and store that preliminary solution.

```
IloSolver solver(mdl);
IloDispatcher dispatcher(solver);
IloRoutingSolution solution(mdl);
IloGoal instantiateCost = IloDichotomize(env,
                                         dispatcher.getCostVar(),
                                         IloFalse);
IloGoal goal = IloInsertionGenerate(env, instantiateCost);
if (!solver.solve(goal)) {
  solver.error() << "Infeasible Routing Plan" << endl;
  exit(-1);
}
solution.store(solver);
```

Once we have stored a preliminary solution to the problem, we then attempt to improve the solution. To do so, we first define the neighborhood that interests us in terms of predefined ILOG Dispatcher moves. Then we use the predefined goal `IloSingleMove` from ILOG Dispatcher and the facilities in ILOG Solver for synchronizing a search with changes in a model.

```
IloNHood nhood = IloTwoOpt(env)
               + IloOrOpt(env)
               + IloRelocate(env)
               + IloExchange(env)
               + IloCross(env);
IloGoal improve = IloSingleMove(env,
                                solution,
                                nhood,
                                IloImprove(env),
                                IloFirstSolution(env),
                                instantiateCost);
while (solver.solve(improve)) {}
```

Once we reach a local minimum, we add the best results to our model.

```
solver.solve(IloRestoreSolution(env, solution));
Info(dispatcher);
```

Displaying a Solution

To display the information that interests us, we define a function `Info`, using many of the predefined features of ILOG Solver and Dispatcher, like this:

```
void Info(IloDispatcher dispatcher) {
  IloSolver solver = dispatcher.getSolver();
  solver.printInformation();
  dispatcher.printInformation();
  solver.out() << "===============" << endl
               << "Cost         : " << dispatcher.getTotalCost() << endl
               << "Number of technicians used : "
               << dispatcher.getNbOfVehiclesUsed() << endl
               << "Solution     : " << endl
               << dispatcher << endl;
}
```

Given a dispatcher (an instance of `IloDispatcher`, documented in the *ILOG Dispatcher Reference Manual*), this user-written function retrieves the presiding solver (an instance of `IloSolver`, documented in the *ILOG Solver Reference Manual*). Then our user-defined function exploits the member function `IloSolver::printInformation` to display generic details about the solver search (such as number of choice points and failures), and `IloDispatcher::printInformation` to display information particular to a VRP, such as number of visits and vehicles.

Ending an Application

To indicate to Concert Technology that our application has completed its use of the environment, we call the member function `IloEnv::end`. This member function cleans up the environment, de-allocating memory used in the model and the search for a solution.

```
env.end();
```

Complete Program

You can see the complete program here or on line in the standard distribution of ILOG Dispatcher in the file `/examples/src/technic1.cpp`. To run this example, you need a license for ILOG Solver and ILOG Dispatcher.

```
#include <ildispat/ilodispatcher.h>

ILOSTLBEGIN

const IloInt NbOfSkills = 3;
```

```
const IloInt NbOfVisitTypes = 10;
const IloInt NbOfTechTypes = 5;

const IloInt Skills[NbOfTechTypes][NbOfSkills] = {{ 0, 2, 5},
                                                  { 3, 0, 4},
                                                  { 2, 3, 2},
                                                  { 4, 2, 1},
                                                  { 4, 4, 0}};

const IloInt RequiredSkills[NbOfVisitTypes][NbOfSkills] = {{ 4, 0, 0},
                                                           { 2, 1, 0},
                                                           { 4, 2, 5},
                                                           { 3, 0, 0},
                                                           { 1, 1, 0},
                                                           { 0, 4, 1},
                                                           { 1, 2, 3},
                                                           { 3, 3, 0},
                                                           { 0, 0, 2},
                                                           { 2, 0, 2}};

void Info(IloDispatcher dispatcher) {
  IloSolver solver = dispatcher.getSolver();
  solver.printInformation();
  dispatcher.printInformation();
  solver.out() << "===============" << endl
               << "Cost         : " << dispatcher.getTotalCost() << endl
               << "Number of technicians used : "
               << dispatcher.getNbOfVehiclesUsed() << endl
               << "Solution     : " << endl
               << dispatcher << endl;
}

int main(int argc, char* argv[]) {
  IloEnv env;
  try {
    IloModel mdl(env);
    IloDimension2 time(env, IloEuclidean, "Time");
    mdl.add(time);
    IloDimension1 skill1(env, "skill1");
    mdl.add(skill1);
    IloDimension1 skill2(env, "skill2");
    mdl.add(skill2);
    IloDimension1 skill3(env, "skill3");
    mdl.add(skill3);
    IloDimension1 skills[NbOfSkills] = {skill1, skill2, skill3};
    ifstream infile;
    char * problem;
    if (argc >=2) problem = argv[1];
    else          problem = (char *) "../../../examples/data/vrp20.dat";
    infile.open(problem);
    if (!infile) {
      env.out() << "File not found or not specified: " << problem << endl;
      env.end();
      return 0;
```

```
}
IloInt nbOfVisits, nbOfTrucks;
infile >> nbOfVisits >> nbOfTrucks;
IloNum capacity;
infile >> capacity;
IloNum depotX, depotY, openTime, closeTime;
infile >> depotX  >> depotY >> openTime >> closeTime;
IloNode depot(env, depotX, depotY);
for (IloInt j = 0; j < nbOfTrucks * NbOfSkills; j++) {
  IloVisit first(depot, "Depot");
  mdl.add(first.getCumulVar(time) >= openTime);
  IloVisit last(depot, "Depot");
  mdl.add(last.getCumulVar(time) <= closeTime);
  char name[16];
  sprintf(name, "Vehicle %ld", j);
  IloVehicle technician(first, last, name);
  technician.setCost(time, 1.0);
  //Skills
  for (IloInt k = 0; k < NbOfSkills; k++) {
    mdl.add(first.getCumulVar(skills[k]) == Skills[j % NbOfTechTypes][k]);
  }
  mdl.add(technician);
}
for (IloInt i = 0; i < nbOfVisits; i++) {
  IloInt id; // visit identifier
  IloNum x, y, quantity, minTime, maxTime, dropTime;
  infile >> id >> x >> y >> quantity >> minTime >> maxTime >> dropTime;
  for (IloInt k = 0; k < NbOfSkills; k++) {
    IloInt requiredSkill = RequiredSkills[i % NbOfVisitTypes][k];
    if (requiredSkill > 0) {
      IloNode customer(env, x, y);
      char name[16];
      sprintf(name, "%ld(%ld)", id, k+1);
      IloVisit visit(customer, name);
      mdl.add(visit.getDelayVar(time) == dropTime);
      mdl.add(minTime <= visit.getCumulVar(time) <= maxTime);
      mdl.add(visit.getCumulVar(skills[k]) >= requiredSkill);
      mdl.add(visit);
    }
  }
}
infile.close();
IloSolver solver(mdl);
IloDispatcher dispatcher(solver);
IloRoutingSolution solution(mdl);
IloGoal instantiateCost = IloDichotomize(env,
                                         dispatcher.getCostVar(),
                                         IloFalse);
IloGoal goal = IloInsertionGenerate(env, instantiateCost);
if (!solver.solve(goal)) {
  solver.error() << "Infeasible Routing Plan" << endl;
  exit(-1);
}
solution.store(solver);
```

```
    IloNHood nhood = IloTwoOpt(env)
                   + IloOrOpt(env)
                   + IloRelocate(env)
                   + IloExchange(env)
                   + IloCross(env);
    IloGoal improve = IloSingleMove(env,
                                    solution,
                                    nhood,
                                    IloImprove(env),
                                    IloFirstSolution(env),
                                    instantiateCost);
    while (solver.solve(improve)) {}
    solver.solve(IloRestoreSolution(env, solution));
    Info(dispatcher);
  } catch(IloException& ex) {
    cerr << "Error: " << ex << endl;
  }
  env.end();
  return 0;
}
```

Results

Here are the results of that program.

```
Number of fails               : 0
Number of choice points       : 0
Number of variables           : 1146
Number of constraints         : 112
Reversible stack (bytes)      : 100524
Solver heap (bytes)           : 537860
Solver global heap (bytes)    : 785380
And stack (bytes)             : 20124
Or stack (bytes)              : 44244
Search Stack (bytes)          : 4044
Constraint queue (bytes)      : 11144
Total memory used (bytes)     : 1503320
Running time since creation   : 0.01
Number of nodes               : 39
Number of visits              : 68
Number of vehicles            : 15
Number of dimensions          : 4
Number of refused moves       : 12491
Number of accepted moves      : 9
Total number of moves         : 12500
===============
Cost         : 813.283
Number of technicians used : 5
Solution     :
Unperformed visits : None
```

```
Vehicle 0 :

-> Depot Time[0..59.6476] skill1[0] skill2[2] skill3[5]
-> 3(1) Time[22.3607..82.0083] skill1[4..Inf) skill2[0..Inf) skill3[0..Inf)
-> 3(2) Time[32.3607..92.0083] skill1[0..Inf) skill2[2..Inf) skill3[0..Inf)
-> 3(3) Time[42.3607..102.008] skill1[0..Inf) skill2[0..Inf) skill3[5..Inf)
-> 12(1) Time[63.541..123.189] skill1[2..Inf) skill2[0..Inf) skill3[0..Inf)
-> 12(2) Time[73.541..133.189] skill1[0..Inf) skill2[1..Inf) skill3[0..Inf)
-> 4(1) Time[149..159] skill1[3..Inf) skill2[0..Inf) skill3[0..Inf)
-> 2(2) Time[179.224..192] skill1[0..Inf) skill2[1..Inf) skill3[0..Inf)
-> 2(1) Time[189.224..202] skill1[2..Inf) skill2[0..Inf) skill3[0..Inf)
-> Depot Time[217.224..230] skill1[0..Inf) skill2[0..Inf) skill3[0..Inf)

Vehicle 1 :

-> Depot Time[0..19.8913] skill1[3] skill2[0] skill3[4]
-> 6(3) Time[11.1803..31.0716] skill1[0..Inf) skill2[0..Inf) skill3[1..Inf)
-> 6(2) Time[21.1803..41.0716] skill1[0..Inf) skill2[4..Inf) skill3[0..Inf)
-> 5(2) Time[41.1803..61.0716] skill1[0..Inf) skill2[1..Inf) skill3[0..Inf)
-> 5(1) Time[51.1803..71.0716] skill1[1..Inf) skill2[0..Inf) skill3[0..Inf)
-> 8(2) Time[95] skill1[0..Inf) skill2[3..Inf) skill3[0..Inf)
-> 8(1) Time[105] skill1[3..Inf) skill2[0..Inf) skill3[0..Inf)
-> 17(3) Time[128.928..145.972] skill1[0..Inf) skill2[0..Inf) skill3[3..Inf)
-> 17(2) Time[138.928..155.972] skill1[0..Inf) skill2[2..Inf) skill3[0..Inf)
-> 17(1) Time[148.928..165.972] skill1[1..Inf) skill2[0..Inf) skill3[0..Inf)
-> 18(2) Time[176.956..194] skill1[0..Inf) skill2[3..Inf) skill3[0..Inf)
-> 18(1) Time[186.956..204] skill1[3..Inf) skill2[0..Inf) skill3[0..Inf)
-> Depot Time[212.768..230] skill1[0..Inf) skill2[0..Inf) skill3[0..Inf)

Vehicle 2 :

-> Depot Time[0..33.0841] skill1[2] skill2[3] skill3[2]
-> 1(1) Time[15.2315..48.3156] skill1[4..Inf) skill2[0..Inf) skill3[0..Inf)
-> 20(3) Time[41.724..74.808] skill1[0..Inf) skill2[0..Inf) skill3[2..Inf)
-> 20(1) Time[51.724..84.808] skill1[2..Inf) skill2[0..Inf) skill3[0..Inf)
-> 9(3) Time[97..105.988] skill1[0..Inf) skill2[0..Inf) skill3[2..Inf)
-> 13(2) Time[159] skill1[0..Inf) skill2[2..Inf) skill3[0..Inf)
-> 13(1) Time[169] skill1[4..Inf) skill2[0..Inf) skill3[0..Inf)
-> Depot Time[190.18..230] skill1[0..Inf) skill2[0..Inf) skill3[0..Inf)

Vehicle 3 :

-> Depot Time[0..20.3246] skill1[4] skill2[2] skill3[1]
-> 10(3) Time[25.4951..45.8197] skill1[0..Inf) skill2[0..Inf) skill3[2..Inf)
-> 10(1) Time[35.4951..55.8197] skill1[2..Inf) skill2[0..Inf) skill3[0..Inf)
-> 11(1) Time[67..77] skill1[4..Inf) skill2[0..Inf) skill3[0..Inf)
-> 19(3) Time[84.0711..156.82] skill1[0..Inf) skill2[0..Inf) skill3[2..Inf)
-> 7(1) Time[105.251..178] skill1[1..Inf) skill2[0..Inf) skill3[0..Inf)
-> 7(2) Time[115.251..188] skill1[0..Inf) skill2[2..Inf) skill3[0..Inf)
-> 7(3) Time[125.251..198] skill1[0..Inf) skill2[0..Inf) skill3[3..Inf)
-> Depot Time[156.465..230] skill1[0..Inf) skill2[0..Inf) skill3[0..Inf)
```

```
Vehicle 4 :

-> Depot Time[0..30.5862] skill1[4] skill2[4] skill3[0]
-> 15(2) Time[61] skill1[0..Inf) skill2[1..Inf) skill3[0..Inf)
-> 15(1) Time[71] skill1[1..Inf) skill2[0..Inf) skill3[0..Inf)
-> 14(1) Time[96.8114..107.204] skill1[3..Inf) skill2[0..Inf) skill3[0..Inf)
-> 16(3) Time[117.992..128.384] skill1[0..Inf) skill2[0..Inf) skill3[1..Inf)
-> 16(2) Time[127.992..138.384] skill1[0..Inf) skill2[4..Inf) skill3[0..Inf)
-> 13(3) Time[159..169] skill1[0..Inf) skill2[0..Inf) skill3[5..Inf)
-> Depot Time[180.18..230] skill1[0..Inf) skill2[0..Inf) skill3[0..Inf)

Vehicle 5 :  Unused
Vehicle 6 :  Unused
Vehicle 7 :  Unused
Vehicle 8 :  Unused
Vehicle 9 :  Unused
Vehicle 10 :  Unused
Vehicle 11 :  Unused
Vehicle 12 :  Unused
Vehicle 13 :  Unused
Vehicle 14 :  Unused
```

C H A P T E R

9

Filling Tankers: Using ILOG Configurator

This chapter presents a problem in fulfilling customer orders to demonstrate configuration features in ILOG Configurator and Concert Technology. The example is based on the idea of configuring tanks on trucks to transport products to satisfy customer orders efficiently. In it, you will learn:

- how to represent a configuration model in terms of its components;
- how to use ports to express relations between components of a model;
- how to use attributes of components in a model;
- how to use goals to search for a configuration solving the problem.

Describing the Problem

For this example, we assume that we have eleven orders to fill for five different products. An order from a customer indicates which of the five products to deliver in which quantities. We will deliver the orders to customers in trucks composed of tanks. We can configure a truck with at most five tanks; there are five types of tanks, each one corresponding to one of the five types of products. The tanks on a truck may be of different capacities; the capacities vary in this way: {1000, 2000, 3000, 4000, 5000}.

There are some limitations on how we configure the tanks on a truck to fulfill the orders:

- We can assign an order to one and only one truck; in other words, we cannot break up an order over several different trucks.
- We must assign different products in an order to different tanks on a truck; that is, we cannot mix different products within a tank.
- We must assign a product from an order to one and only one tank.
- We can, however, assign the same product from different orders to a given tank on a given truck.
- Trucks have a maximum capacity of 12000.
- We must not exceed the stated capacity of a tank.
- A truck cannot carry more than three different types of products.
- Two of the products—P3 and P4—are incompatible; it is not possible to transport them on the same truck.

Initially, we do not know how many trucks and tanks we will need to fill the orders. There is no limit on those numbers. A solution of the problem consists of generating appropriate sets of tanks and trucks to fill all the orders.

Representing the Data

How will ILOG Configurator help us represent this problem? Configurator offers facilities for representing the four basic *components* of the problem: orders, products, tanks, trucks. In Configurator, the components may also have *attributes* (such as quantity, capacity, and so forth) and *ports*. Ports represent relations between components; they may be either HasPart ports or Uses ports. Configurator provides classes and functions to implement these relations. (For a more detailed explanation of ports, see the *ILOG Configurator Reference* and *User's Manuals*.) Figure 9.1 illustrates the Configurator components with their attributes and ports as we will use them in our model.

In particular, as you see in Figure 9.1, Order components and Product components have the attribute of quantity. In our model, the quantity of an order indicates the sum of quantities of the products in that order. Truck components also have an attribute of quantity: their load, indicating the quantity of products that a truck carries.

As components, tanks have two attributes: one indicates the capacity of the tank; the other shows the quantity of product the tank is assigned to carry.

Besides their attributes, the components also have ports. On an Order component, for example, there is a HasPart port that relates the order to the products in that order. There is also a Uses port on an Order component, relating the order to the truck that will fulfill the order. Reciprocally, a Truck component includes a Uses port relating it to the order(s) that it

fills. There are similar reciprocal Uses ports relating Product components to the tanks where they will be carried and relating the Tank components to the products they will contain.

To represent the tanks that will compose a truck, the Truck components include a HasPart port relating the truck to the tanks on it.

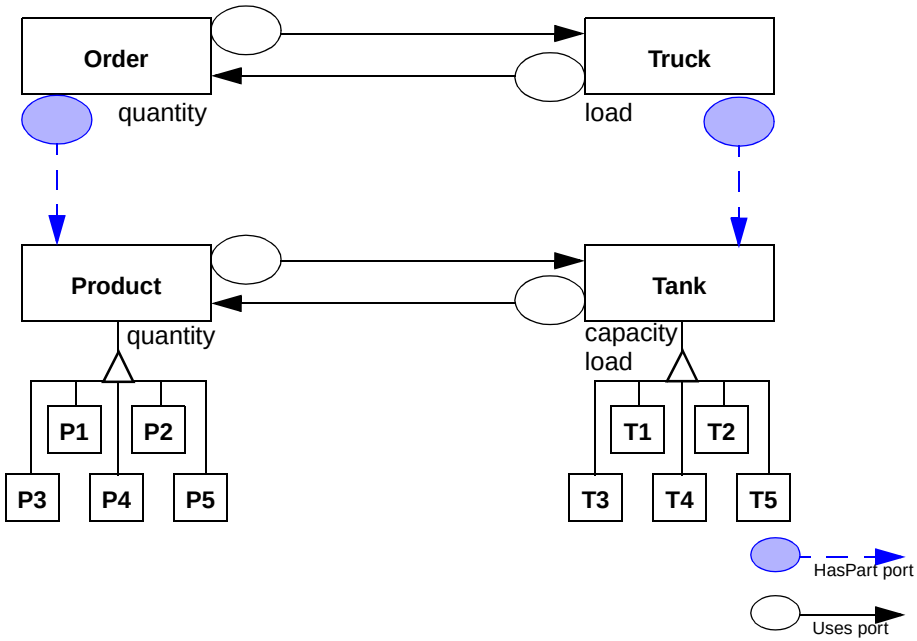

Figure 9.1 *A Model for Filling Tankers*

Developing a Model

How shall we implement that model of the problem? First, we create an environment (an instance of `IloEnv`, documented in the *ILOG Concert Technology Reference Manual*) for the model. Then we open a `try {}` to catch any exceptions that may arise. Next, because this is a clearly a configuration problem, we define a catalog, an instance of `IloCatalog`, documented in the *ILOG Configurator Reference Manual*. The catalog contains the descriptions of the types of components that can be used in the final configuration.

```
IloEnv env;
try {
  IloCatalog cat(env, “Tankers”);
```

A component type is a generic description of a category of components. This description consists of a set of attributes, a set of ports, and a set of constraints on these attributes and ports. Each instance of a type has its own copy of the attributes, ports, and constraints defined for this type or for its ancestors.

Now we need to represent the types of components of the model, in particular, the products, the orders, the tanks, and the trucks. We also need to represent the constraints on and among the components, such constraints as capacity of tanks, compatibility between products, and so forth.

Representing Products

As a first step in representing products in our model, we add `Product` as an abstract component type in our catalog. Then, for that component type, we add `quantity` as an integer attribute of a product. Finally, for this particular problem, we define the concrete types of products (that is, the product types `P1`, `P2`, `P3`, and so forth) as subtypes of `Product`. All these subtypes will inherit the attribute `quantity` defined for `Product`.

```
IloComponentType Product = cat.addType("Product");
Product.addIntAttr("quantity", 1, 100000);

IloComponentType P1 = cat.addType("P1", "Product");
IloComponentType P2 = cat.addType("P2", "Product");
IloComponentType P3 = cat.addType("P3", "Product");
IloComponentType P4 = cat.addType("P4", "Product");
IloComponentType P5 = cat.addType("P5", "Product");
```

Representing Orders

As we did for products, we first add `Order` as a component type in our catalog. We observe that the quantity of an order is the sum of quantities of products in the order. We also note that an order is made up of one to three different product specifications. To represent that idea, we define a port, `oprod`, as the products in the order, and we constrain the quantity of the order to be the sum of quantities of products in the order, like this:

```
IloComponentType Order = cat.addType("Order");

IloNumAttr  q = Order.addIntAttr("quantity", 1, 100000);
IloPort oprod = Order.addPort(IloHasPart,
                              Product,
                              "products", 1, 3,
                              IloEntry);

Order.add(q == oprod.getSum("quantity"));
```

Representing Tanks

Again, when we want to represent a component in our model, we begin by adding that component type (in this case, Tank) to our catalog. Next, we add the attributes of that component type. For the capacity of a tank, we use an integer attribute. For its load (the amount of a product actually put into a tank), we also use an integer attribute.

Then to relate the product to a tank, we define a port `tprod`, indicating that a product is assigned to a tank. We use the predefined function `IloInversePort` to indicate that a tank and product are inversely related; that is, if a tank contains a product, then that product is assigned to that tank.

We constrain the load of a tank `tload` to the be the sum of the quantities of the products in that tank. We also constrain `tload` to be less than the capacity of the tank.

Then for this particular problem, we define the available types of tanks (`T1`, `T2`, `T3`, and so forth) that can be used in the final configuration.

```
IloNumArray capacities(env, 5, 1000, 2000, 3000, 4000, 5000);

IloComponentType Tank = cat.addType(“Tank”);

IloNumAttr capa  = Tank.addIntAttr(“capacity”, capacities);
IloNumAttr tload = Tank.addIntAttr(“load”, 0, 5000);
IloPort    tprod = Tank.addPort(IloUses,
                                Product,
                                “products”,
                                0, 5000);
IloExclusivePort(Tank, “products”);
IloInversePort(Tank, “products”, Product, “tank”);

Product.addPort(IloUses, Tank, “tank”, 1);

Tank.add( tload == tprod.getSum(“quantity”) );
Tank.add( tload <= capa );

IloComponentType T1   = cat.addType(“T1”, “Tank”);
IloComponentType T2   = cat.addType(“T2”, “Tank”);
IloComponentType T3   = cat.addType(“T3”, “Tank”);
IloComponentType T4   = cat.addType(“T4”, “Tank”);
IloComponentType T5   = cat.addType(“T5”, “Tank”);
```

Representing Trucks

By now you know the routine: we begin by adding the component type `Truck` to the catalog. Then we define a port, an instance of `IloPort`, to represent the set of orders assigned to a truck.

Next we use the predefined function `IloInversePort` to indicate that a truck and order are inversely related. That is, we stipulate that if a truck carries an order, then the order is assigned to that truck.

We must also declare that a truck is composed of one to five tanks. We do so with the port `tanks`, an instance of `IloHasPart`.

We define load as a numeric attribute of a truck, and we represent the overall load of a truck as less than or equal to 12000. We also constrain the load to be the sum of the loads in the tanks on the truck. At the same time, the load of a truck must be the sum of the quantities in the orders using that truck.

To calculate which products will be on the truck, we check the ports of the `orders` and the `tanks` on the truck.

```
IloComponentType Truck   = cat.addType(“Truck”);
IloPort orders = Truck.addPort(IloUses,
                               Order,
                               “orders”,
                               1, 100000);
IloExclusivePort(Truck, “orders”);

Order.addPort(IloUses, Truck, “truck”, 1);
IloInversePort(Order, “truck”, Truck, “orders”);

IloPort tanks  = Truck.addPort(IloHasPart,
                               Tank,
                               “tank”,
                               1, 5);

IloNumAttr load = Truck.addIntAttr(“load”, 1, 100000);

Truck.add( load <= 12000 );
Truck.add( load == orders.getSum(“quantity”) );
Truck.add( load == tanks .getSum(“load”) );

tprod = orders.getPortUnion(“products”);
Truck.add( tprod == tanks.getPortUnion(“products”) );
```

All the tanks on a given truck have different capacities. The predefined member function `getNumUnion`, documented in the *ILOG Configurator Reference Manual*, returns the set of all the different capacities of the tanks on a truck. We specify that each tank of a truck has a different capacity by constraining the number of capacity values to be equal to the number of tanks, like this:

```
Truck.add( IloCard(tanks.getNumAttr(“capacity”))
           ==
           IloCard(tanks) );
```

Our problem allows no more than three different types of products on any given truck. The port `tprod` contains the union of all products assigned to the tanks of a truck. The

predefined member function `getCardOf`, documented in the *ILOG Configurator Reference Manual*, returns the number of components connected to `tprod` and specialized for the given type of product.

```
IloConstraint cP1 = (tprod.getCardOf(P1) > 0);
IloConstraint cP2 = (tprod.getCardOf(P2) > 0);
IloConstraint cP3 = (tprod.getCardOf(P3) > 0);
IloConstraint cP4 = (tprod.getCardOf(P4) > 0);
IloConstraint cP5 = (tprod.getCardOf(P5) > 0);

Truck.add( cP1 + cP2 + cP3 + cP4 + cP5 <= 3 );
```

We also recall that in our problem, two products `P3` and `P4`, are incompatible. Our solution must not put those products on the same truck, so we state that constraint like this:

```
Truck.add( cP3 + cP4 <= 1 );
```

To guarantee that products are compatible with tanks in a configuration, we use a type table, an instance of `IloTypeTable`, documented in the *ILOG Configurator Reference Manual*. A type table concisely defines the compatibility (or incompatibilities) between component types in a catalog. In our case, we are interested in the compatibilities between products and tanks. The predefined ILOG Configurator function `IloTypeCompatibility` offers a straightforward way to use the information in a type table, like this:

```
IloTypeTable tt = cat.addTypeTable(Product, Tank, IloCompatible);
tt.add(P1, T1);
tt.add(P2, T2);
tt.add(P3, T3);
tt.add(P4, T4);
tt.add(P5, T5);

Tank.add( IloTypeCompatibility( Tank.getTypeAttr(),
                                Tank.getPort("products"),
                                tt ) );
```

We define a particular type of component `Tankers` that represents a set of orders fulfilled by a set of trucks. An instance of `Tankers` contains two has-part ports; one port represents the set of orders; the other represents the set of trucks. A constraint stipulates that the union of orders carried by the trucks is equal to the set of orders in the instance of `Tankers`.

```
IloComponentType Tankers = cat.addType("Tankers");
Tankers.addPort(IloHasPart, Order, "orders", IloEntry);
Tankers.addPort(IloHasPart, Truck, "trucks");
Tankers.add( Tankers.getPort("orders")
             ==
             Tankers.getPort("trucks").getPort("orders"));
```

Defining the Configuration Request

So far, we have worked on the generic and structural aspects of our problem. That is, we have defined the component *types* in our catalog, and stated the structural and generic constraints between these component types. Moving beyond this generic information in the catalog (an instance of `IloCatalog`), we are ready now to define a configuration, an instance of `IloConfiguration`, documented in the *ILOG Configurator Reference Manual*. A configuration is an instance of `IloModel` (documented in the *ILOG Concert Technology Reference Manual*) that contains an initial set of components on which additional specific constraints may be added. In short, a configuration makes the generic structural information of a catalog concrete and specific. In our example, the initial request consists of an instance of the type Tankers with its set of orders and product specifications.

```
IloConfiguration request(cat, “Tankers”);
IloComponent tankers = request.makeInstance(Tankers, “Tankers”);
request.close(Tankers);
MakeOrders(request, tankers, Order, Product);
```

The function `MakeOrders` is one we wrote for this particular problem. We define it in this way:

```
void MakeOrders(IloConfiguration request,
                IloComponent tankers,
                IloComponentType Order,
                IloComponentType Product) {
  char buffer[40];
  IloCatalog cat = request.getCatalog();
  IloPort ptk = tankers.getPort(“orders”);
  for(IloInt i=0; i < NbOrders; i++) {
    IloInt id = i+1;
    sprintf(buffer,”o%d”, id);
    IloComponent c = request.makeInstance(Order, buffer);
    ptk.connect(c);
    IloPort prods = c.getPort(“products”);
    IloInt q;
    IloComponent p;
    //P1
    q = Quantities[i][0];
    if (q > 0) {
      sprintf(buffer, “o%d.P1”, id);
      p = request.makeInstance(cat.getType(“P1”), buffer);
      p.getNumAttr(“quantity”).setValue(q);
      prods.connect( p );
    }

    //P2
    q = Quantities[i][1];
    if (q > 0) {
      sprintf(buffer, “o%d.P2”, id);
```

```
      p = request.makeInstance(cat.getType("P2"), buffer);
      p.getNumAttr("quantity").setValue(q);
      prods.connect( p );
    }

    //P3
    q = Quantities[i][2];
    if (q > 0) {
      sprintf(buffer, "o%d.P3", id);
      p = request.makeInstance(cat.getType("P3"), buffer);
      p.getNumAttr("quantity").setValue(q);
      prods.connect( p );
    }

    //P4
    q = Quantities[i][3];
    if (q > 0) {
      sprintf(buffer, "o%d.P4", id);
      p = request.makeInstance(cat.getType("P4"), buffer);
      p.getNumAttr("quantity").setValue(q);
      prods.connect( p );
    }

    //P5
    q = Quantities[i][4];
    if (q > 0) {
      sprintf(buffer, "o%d.P5", id);
      p = request.makeInstance(cat.getType("P5"), buffer);
      p.getNumAttr("quantity").setValue(q);
      prods.connect( p );
    }
    prods.close();
  }
  request.close(Order);
  request.close(Product);
}
```

Solving the Problem

We search for a preliminary solution in this way:

```
IloSolver solver(env);
IloConfigurator cfg(solver);
solver.extract(request);
```

We use that first solution as an upper bound on the number of trucks needed.

We then formulate our chief goal: to configure orders and products. We express that goal in this way:

```
IloGoal mainGoal = MainGoal(env, cfg, Order, Product);
```

As long as there are orders to fill, we continue to search.

```
if (solver.solve(mainGoal)) {
  IloInt nbTrucks = cfg.getNumberOfInstances(Truck);
  solver.out() << "** First solution at " << nbTrucks << " trucks."
               << endl;

  IloNumVar costs = IloCard(tankers.getPort("trucks"));
  IloNumVar negCosts(env, -nbTrucks, 0, ILOINT);
  request.add( negCosts == - costs );
  request.add( IloMinimize(env, costs) );

  solver.solve( IloInstantiate(env, negCosts) && mainGoal );
```

Displaying a Solution

To display the solution, we iterate over the set of trucks in the instance `tankers` and recursively display the set of tanks on each truck.

```
  solver.out() << "---------------------------------------------------"
               << endl
               << "   Optimum at " << cfg.getNumberOfInstances(Truck)
               << " trucks." << endl
               << "---------------------------------------------------"
               << endl << endl;
  PrintTrucks(solver.out(), cfg, Truck);
  solver.printInformation();
```

Ending an Application

To indicate to Concert Technology that our application has completed its use of the environment, we call the member function `IloEnv::end`. This member function cleans up the environment, de-allocating memory used in the model and the search for a solution.

```
env.end();
```

Complete Program

You can see the complete program here or on line in the standard distribution of ILOG Configurator in the file `/examples/src/tankers.cpp`. To run this example yourself, you need a license for ILOG Configurator version 2.1 plus ILOG Solver version 5.1.

```
// ------------------------------------------------------------ -*- C++ -*-
// File: examples/src/tankers.cpp
// ------------------------------------------------------------------------

/*------------------------------------------------------------*/
/* Copyright (C) 1999-2001 by ILOG.                           */
/* All Rights Reserved.                                       */
/* Permission is expressly granted to use this example in the */
/* course of developing applications that use ILOG products.  */
/*------------------------------------------------------------*/

#include <ilconfig/iloconfig.h>
#include <ilconfig/ilcconfig.h>
ILOSTLBEGIN

//-----------------------------------------------------
// Data
//-----------------------------------------------------
const IloInt NbProducts =  5;
const IloInt NbOrders   = 11;

const IloInt Quantities[NbOrders][NbProducts] = {
  { 1000,    0,  500,    0, 2000 },
  {    0, 2500,    0,  500, 1000 },
  { 2000, 3000, 5000,    0,    0 },
  { 1000,  500,    0,    0,    0 },
  {    0, 3000, 2000,    0, 4000 },
  {    0,    0,    0,    0, 3000 },
  {    0,  500,  500,    0,    0 },
  {    0,  500,    0, 2000,  500 },
  {    0, 1000,    0,    0, 1000 },
  { 2000,    0,    0,    0, 4000 },
  { 1500,    0,    0,    0,    0 }
};

//-----------------------------------------------------
void MakeOrders(IloConfiguration request,
                IloComponent tankers,
                IloComponentType Order,
                IloComponentType Product) {
  char buffer[40];
  IloCatalog cat = request.getCatalog();
  IloPort ptk = tankers.getPort("orders");
  for(IloInt i=0; i < NbOrders; i++) {
    IloInt id = i+1;
```

```
      sprintf(buffer,"o%d", id);
      IloComponent c = request.makeInstance(Order, buffer);
      ptk.connect(c);
      IloPort prods = c.getPort("products");
      IloInt q;
      IloComponent p;
      //P1
      q = Quantities[i][0];
      if (q > 0) {
        sprintf(buffer, "o%d.P1", id);
        p = request.makeInstance(cat.getType("P1"), buffer);
        p.getNumAttr("quantity").setValue(q);
        prods.connect( p );
      }

      //P2
      q = Quantities[i][1];
      if (q > 0) {
        sprintf(buffer, "o%d.P2", id);
        p = request.makeInstance(cat.getType("P2"), buffer);
        p.getNumAttr("quantity").setValue(q);
        prods.connect( p );
      }

      //P3
      q = Quantities[i][2];
      if (q > 0) {
        sprintf(buffer, "o%d.P3", id);
        p = request.makeInstance(cat.getType("P3"), buffer);
        p.getNumAttr("quantity").setValue(q);
        prods.connect( p );
      }

      //P4
      q = Quantities[i][3];
      if (q > 0) {
        sprintf(buffer, "o%d.P4", id);
        p = request.makeInstance(cat.getType("P4"), buffer);
        p.getNumAttr("quantity").setValue(q);
        prods.connect( p );
      }

      //P5
      q = Quantities[i][4];
      if (q > 0) {
        sprintf(buffer, "o%d.P5", id);
        p = request.makeInstance(cat.getType("P5"), buffer);
        p.getNumAttr("quantity").setValue(q);
        prods.connect( p );
      }
      prods.close();
    }
    request.close(Order);
```

```
  request.close(Product);
}

//--------------------------------------------------
void PrintTrucks(ILOSTD(ostream)& os, IloConfigurator cfg, IloComponentType t);
void PrintTank  (ILOSTD(ostream)& os, IloConfigurator cfg, IloComponent c);

//--------------------------------------------------
// Strategy: choose first the biggest order
//--------------------------------------------------
class BiggestOrderFirstI : public IloCPInstanceSelectI {
public:
  BiggestOrderFirstI(IloConfigurator cfg):IloCPInstanceSelectI(cfg) {}
  IloComponent select(IloComponentType t);
};
IloComponent BiggestOrderFirstI::select(IloComponentType t) {
  IloComponent best;
  IloNum q=0;
  IloConfigurator cfg = getConfigurator();
  for(IloInstanceIterator it(cfg, t); it.ok(); ++it) {
    IloComponent c = *it;
    if (! cfg.isConfigured(c)) {
      IloNum eval = c.getNumAttr("quantity").getUb();
      if (eval > q) {
        q = eval;
        best = c;
      }
    }
  }
  return best;
}
IloCPInstanceSelect BiggestOrderFirst(IloConfigurator cfg) {
  return new (cfg.getHeap()) BiggestOrderFirstI(cfg);
}

ILO_CONFIG_STRATEGY0(Product_Solver) {
  return instantiate(getComponent().getPort("tank"), IlcFalse);
}
ILO_CONFIG_STRATEGY0(Order_Solver) {
  IloComponent c = getComponent();
  return IlcAnd(instantiate(c.getPort("truck"), IlcFalse),
                maxInstantiate(c.getPort("products"))
                );
}

ILCGOAL3(IlcMainGoal, IloConfigurator, cfg,
         IloComponentType, Order, IloComponentType, Product) {
  cfg.setStrategy(Order,   Order_Solver  (cfg));
  cfg.setStrategy(Product, Product_Solver(cfg));
  return cfg.configure( Order, BiggestOrderFirst(cfg) );
}
ILOCPGOALWRAPPER3(MainGoal, IloConfigurator, cfg,
```

```
                    IloComponentType, Order, IloComponentType, Product) {
  return IlcMainGoal(solver, cfg, Order, Product);
}

//--------------------------------------------------
// Main program
//--------------------------------------------------
int main(int, char* []) {
  IloEnv env;
  try {
    IloCatalog cat(env, "Tankers");

    //Product model
    IloComponentType Product = cat.addType("Product");
    Product.addIntAttr("quantity", 1, 100000);

    IloComponentType P1 = cat.addType("P1", "Product");
    IloComponentType P2 = cat.addType("P2", "Product");
    IloComponentType P3 = cat.addType("P3", "Product");
    IloComponentType P4 = cat.addType("P4", "Product");
    IloComponentType P5 = cat.addType("P5", "Product");

    //Order model
    IloComponentType Order = cat.addType("Order");

    IloNumAttr  q = Order.addIntAttr("quantity", 1, 100000);
    IloPort oprod = Order.addPort(IloHasPart,
                                  Product,
                                  "products", 1, 3,
                                  IloEntry);

    Order.add(q == oprod.getSum("quantity"));


    // Tank model
    IloNumArray capacities(env, 5, 1000, 2000, 3000, 4000, 5000);

    IloComponentType Tank = cat.addType("Tank");

    IloNumAttr capa  = Tank.addIntAttr("capacity", capacities);
    IloNumAttr tload = Tank.addIntAttr("load", 0, 5000);
    IloPort    tprod = Tank.addPort(IloUses,
                                    Product,
                                    "products",
                                    0, 5000);
    IloExclusivePort(Tank, "products");
    IloInversePort(Tank, "products", Product, "tank");

    Product.addPort(IloUses, Tank, "tank", 1);

    Tank.add( tload == tprod.getSum("quantity") );
    Tank.add( tload <= capa );
```

```
IloComponentType T1   = cat.addType("T1", "Tank");
IloComponentType T2   = cat.addType("T2", "Tank");
IloComponentType T3   = cat.addType("T3", "Tank");
IloComponentType T4   = cat.addType("T4", "Tank");
IloComponentType T5   = cat.addType("T5", "Tank");

// Truck model
IloComponentType Truck   = cat.addType("Truck");
IloPort orders = Truck.addPort(IloUses,
                               Order,
                               "orders",
                               1, 100000);
IloExclusivePort(Truck, "orders");

Order.addPort(IloUses, Truck, "truck", 1);
IloInversePort(Order, "truck", Truck, "orders");

IloPort tanks  = Truck.addPort(IloHasPart,
                               Tank,
                               "tank",
                               1, 5);

IloNumAttr load = Truck.addIntAttr("load", 1, 100000);

Truck.add( load <= 12000 );
Truck.add( load == orders.getSum("quantity") );
Truck.add( load == tanks .getSum("load") );

tprod = orders.getPort("products");
Truck.add( tprod == tanks.getPort("products") );

// All tanks have different capacities
Truck.add( IloCard(tanks.getNumAttr("capacity"))
           ==
           IloCard(tanks) );

// Not more than three different type of products
IloConstraint cP1 = (tprod.getCardOf(P1) > 0);
IloConstraint cP2 = (tprod.getCardOf(P2) > 0);
IloConstraint cP3 = (tprod.getCardOf(P3) > 0);
IloConstraint cP4 = (tprod.getCardOf(P4) > 0);
IloConstraint cP5 = (tprod.getCardOf(P5) > 0);

Truck.add( cP1 + cP2 + cP3 + cP4 + cP5 <= 3 );

// P3, P4 incompatibility
Truck.add( cP3 + cP4 <= 1 );

//Product, Tank compatibility
IloTypeTable tt = cat.addTypeTable(Product, Tank, IloCompatible);
tt.add(P1, T1);
tt.add(P2, T2);
```

```
tt.add(P3, T3);
tt.add(P4, T4);
tt.add(P5, T5);

Tank.add( IloTypeCompatibility( Tank.getTypeAttr(),
                                Tank.getPort("products"),
                                tt ) );

//Tankers model
IloComponentType Tankers = cat.addType("Tankers");
Tankers.addPort(IloHasPart, Order, "orders", IloEntry);
Tankers.addPort(IloHasPart, Truck, "trucks");
Tankers.add( Tankers.getPort("orders")
             == Tankers.getPort("trucks").getPort("orders") );

if (! cat.build()) {
  env.out() << "Error: Bad Catalog." << endl;
  env.end();
  return 0;
}
//Instances
IloConfiguration request(cat, "Tankers");
IloComponent tankers = request.makeInstance(Tankers, "Tankers");
request.close(Tankers);
MakeOrders(request, tankers, Order, Product);

// Search for first solution (upper bound on number of trucks)
IloSolver solver(env);
IloConfigurator cfg(solver);
solver.extract(request);

IloGoal mainGoal = MainGoal(env, cfg, Order, Product);
if (solver.solve(mainGoal)) {
  IloInt nbTrucks = cfg.getNumberOfInstances(Truck);
  solver.out() << "** First solution at " << nbTrucks << " trucks."
               << endl;

  IloNumVar costs = IloCard(tankers.getPort("trucks"));
  IloNumVar negCosts(env, -nbTrucks, 0, ILOINT);
  request.add( negCosts == - costs );
  request.add( IloMinimize(env, costs) );

  solver.solve( IloInstantiate(env, negCosts) && mainGoal );

  solver.out() << "-------------------------------------------------"
               << endl
               << "   Optimum at " << cfg.getNumberOfInstances(Truck)
               << " trucks." << endl
               << "-------------------------------------------------"
               << endl << endl;
  PrintTrucks(solver.out(), cfg, Truck);
  solver.printInformation();
}
```

```
    else {
      solver.out() << "-------------------------" << endl
                   << "   No solution found ! "   << endl
                   << "-------------------------" << endl;
    }
  }
  catch (IloException& ex) {
    cerr << "Error: " << ex << endl;
  }
  env.end();
  return 0;
}

//------------------------------------------------------
// Display
//------------------------------------------------------
void PrintTank(ILOSTD(ostream)& os, IloConfigurator cfg, IloComponent tank) {
  os << "       <" << tank.getName() << ">" << endl
     << "              --> Type      = "
     << cfg.getTypeAttr(tank.getTypeAttr()).getCurrentType().getName()
     << endl
     << "              --> Load      = "
     << cfg.getIntAttr(tank.getNumAttr("load")) << endl
     << "              --> Products  = "
     << cfg.getPort(tank.getPort("products")) << endl
     << endl;
}

void PrintTrucks(ILOSTD(ostream)& os, IloConfigurator cfg,
                 IloComponentType Truck) {
  os << "##### TRUCKS (" << cfg.getNumberOfInstances(Truck)
     << ") #####" << endl;
  for(IloInstanceIterator it(cfg, Truck); it.ok(); ++it) {
    IloComponent truck = *it;
    IlcPort orders = cfg.getPort(truck.getPort("orders"));
    IlcPort tanks  = cfg.getPort(truck.getPort("tank"));

    os << "<Truck " << truck.getName() << ">" << endl
       << "       --> Load   = "
       << cfg.getIntAttr(truck.getNumAttr("load")) << endl
       << "       --> Orders = " << orders << endl
       << endl;
    os << "       --> Tanks  = " << tanks << endl;
    for(IlcConnectionIterator itt2(tanks, IlcRequiredIteration);
        itt2.ok(); ++itt2)
      PrintTank(os, cfg, *itt2);
  }
}
```

Results

Here are the results of that program.

```
** First solution at 5 trucks.
?? Try with 4 trucks.
?? Try with 3 trucks.
-------------------------------------------------
   Optimum at 4 trucks.
-------------------------------------------------

##### TRUCKS [4] #####
<Truck Truck#1>
       --> Load   = [11000]
       --> Orders = [ o1 o10 o11] ([3])

       <Tank#1>
              --> Type      = T1
              --> Load      = [4500]
              --> Products  = [ o1.P1 o10.P1 o11.P1] ([3])

       <Tank#5>
              --> Type      = T3
              --> Load      = [500]
              --> Products  = [ o1.P3] ([1])

       <Tank#8>
              --> Type      = T5
              --> Load      = [2000]
              --> Products  = [ o1.P5] ([1])

       <Tank#10>
              --> Type      = T5
              --> Load      = [4000]
              --> Products  = [ o10.P5] ([1])

<Truck Truck#2>
       --> Load   = [12000]
       --> Orders = [ o2 o6 o8 o9] ([4])

       <Tank#2>
              --> Type      = T2
              --> Load      = [4000]
              --> Products  = [ o2.P2 o8.P2 o9.P2] ([3])

       <Tank#7>
              --> Type      = T4
              --> Load      = [2500]
              --> Products  = [ o2.P4 o8.P4] ([2])

       <Tank#13>
```

```
                --> Type      = T5
                --> Load      = [4500]
                --> Products  = [ o2.P5 o6.P5 o8.P5] ([3])

        <Tank#14>
                --> Type      = T5
                --> Load      = [1000]
                --> Products  = [ o9.P5] ([1])

<Truck Truck#3>
        --> Load   = [11500]
        --> Orders = [ o3 o4] ([2])

        <Tank#3>
                --> Type      = T2
                --> Load      = [3500]
                --> Products  = [ o3.P2 o4.P2] ([2])

        <Tank#6>
                --> Type      = T3
                --> Load      = [5000]
                --> Products  = [ o3.P3] ([1])

        <Tank#11>
                --> Type      = T1
                --> Load      = [3000]
                --> Products  = [ o3.P1 o4.P1] ([2])

<Truck Truck#4>
        --> Load   = [10000]
        --> Orders = [ o5 o7] ([2])

        <Tank#4>
                --> Type      = T2
                --> Load      = [3500]
                --> Products  = [ o5.P2 o7.P2] ([2])

        <Tank#9>
                --> Type      = T5
                --> Load      = [4000]
                --> Products  = [ o5.P5] ([1])

        <Tank#12>
                --> Type      = T3
                --> Load      = [2500]
                --> Products  = [ o5.P3 o7.P3] ([2])

Number of fails               : 0
Number of choice points       : 16
Number of variables           : 613
Number of constraints         : 981
Reversible stack (bytes)      : 112584
```

```
Solver heap (bytes)           : 574884
Solver global heap (bytes)    : 4044
And stack (bytes)             : 4044
Or stack (bytes)              : 32184
Search Stack (bytes)          : 4044
Constraint queue (bytes)      : 11144
Total memory used (bytes)     : 742928
Elapsed time since creation   : 0.047
```

C H A P T E R

10

Pareto-Optimization: Using ILOG Scheduler

This chapter presents a ship loading problem to demonstrate Pareto-optimization with ILOG Scheduler in Concert Technology. In it, you will learn:

- how to represent activities and resources in a scheduling problem;
- how to use load-leveling to manage demands for resources;
- how to use Pareto-optimal trade-offs to solve a scheduling problem.

This example is the capstone in a series of examples based on the well known ship loading problem of operations research. The simple introductory version of that problem appears in the *ILOG Scheduler User's Manual*, along with implementations of solutions to the gradually more complicated versions of that problem leading up to this example.

What Is Pareto-Optimization?

In addition to the general purpose modeling classes available in Concert Technology, ILOG Scheduler offers a C++ library of classes to represent activities, resources, capacity constraints, and temporal constraints specific to scheduling and resource allocation.

In general terms, scheduling assigns *activities* to *resources* over *time*. Some scheduling problems also manage minimal or maximal *capacity* constraints over time. In many

scheduling problems, the aim is to minimize a temporal criterion: the *makespan*, the overall duration of the schedule. In contrast, in some problems, we must not only determine the minimal makespan, but also avoid peaks in the use of resources over time. In other words, there may be load-leveling considerations in those problems. In those problems, there are at least two optimization criteria to consider: makespan (C_1) and peak resource utilization (C_2). Pareto-optimization is one way of managing such a situation.

In a problem with two optimization criteria, C_1 and C_2, to minimize, a solution S to the problem is *Pareto-optimal* if and only if for any solution S', either $C_1(S)$ is less than or equal to $C_1(S')$ or $C_2(S)$ is less than or equal to $C_2(S')$. In other words, a solution is Pareto-optimal if and only if there is no way to improve it with respect to one criterion without worsening it in terms of the other criterion.

Describing the Problem

In this chapter, we consider a ship loading problem, one in which there are 34 activities, and all of which require resources. Certain of the 34 activities must precede others, as indicated in *Table 10.1, "Activities and Successors."*

Table 10.1 *Activities and Successors*

Act.	Successors	Act.	Successors	Act.	Successors	Act.	Successors
1	2, 4	11	13	21	22	31	28
2	3	12	13	22	23	32	33
3	5, 7	13	15, 16	23	24	33	34
4	5	14	15	24	25	34	
5	6	15	18	25	26, 30, 31, 32		
6	8	16	17	26	27		
7	8	17	18	27	28		
8	9	18	19, 20, 21	28	29		
9	10, 14	19	23	29			
10	11, 12	20	23	30	28		

In formal terms, this is a scheduling problem in which activities require discrete resources of capacity strictly greater than one. In this context, two activities that require the same resource may overlap in time, but the total capacity required at a given time must not exceed the capacity available at that time.

The capacity of a resource is not fixed in this problem. (For a simpler version of the ship-loading problem in which resource capacity is fixed and known in advance, see the *ILOG Scheduler User's Manual*.) In fact, the point of this example is that we can make trade-offs between time and capacity. In concrete terms, we might want to balance the number of resources (say, dockworkers hired) against the amount of time (how many hours, for example) it takes to load the ship.

In solving the problem, we will compute the Pareto-optimal trade-offs between the makespan and resource capacity. As the problem under consideration is rather small (only 34 activities), we can afford to compute all the Pareto-optimal trade-offs exactly. For a larger problem, we might have to use approximations instead, such approximations as optimizing each criterion up to a given precision.

A Preview

Just to give you a quick overview of where we are going, here is a skeleton program that assumes we have already designed a model for the problem and defined goals to guide the search for a solution. This piece of code then finds all the Pareto-optimal solutions to that model (extracted for an instance of `IloSolver`) with respect to two criteria, `X1` and `X2`.

```
IloSolver solver(model);

IloObjective minX1Obj = IloMinimize(X1);
IloObjective minX2Obj = IloMinimize(X2);
IloGoal minX1Goal = ... // Goal to minimize X1;
IloGoal minX2Goal = ... // Goal to minimize X2;
IloInt V1;
IloInt V2 = IloIntMax;
IloConstraint minX2Ct = (X2 < V2);

while (IloTrue) {
  model.add(minX2Ct);
  model.add(minX1Obj);

  if (solver.solve(minX1Goal))
    V1 = solver.getIntVar(X1).getValue();
  else
    break;

  model.remove(minX1Obj);
  model.add(minX2Obj);

  IloConstraint isoX1Ct = (X1 == V1);
  model.add(isoX1Ct);

  if (solver.solve(minX2Goal)) {
    V2 = solver.getIntVar(X2).getValue();
```

```
    solver.out() << "NEW PARETO OPTIMAL SOLUTION" << endl;
    solver.out() << "X1 = " << V1 << endl;
    solver.out() << "X2 = " << V2 << endl;
  } else
    break;

  model.remove(isoX1Ct);
  model.remove(minX2Ct);
  model.remove(minX2Obj);
  minX2Ct = (X2 < V2);
}
```

In the following sections, we will walk through the steps to design the complete model, define appropriate goals, and solve the problem in detail.

Developing a Model

To develop a model for this problem, we write a function, `DefineModel`, that returns an instance of `IloModel` (documented in the *ILOG Concert Technology Reference Manual*). This user-defined function accepts an environment, the number of activities in the problem, their durations, the resources they demand, the order among activities (that is, which activities precede each other). In the model that it returns, the function does several things.

- It computes an upper bound on the horizon of the schedule. It also computes the maximal capacity used. In computing the horizon and capacity of the schedule, we use the operator += from Concert Technology.

```
IloInt horizon = 0;
IloInt capacity = 0;
IloInt i;
IloNum demandMax = 0;
for (i = 0; i < numberOfActivities; i++) {
  horizon += durations[i];
  capacity += demands[i];
  demandMax = IloMax(demandMax, demands[i]);
}
```

- It creates a makespan variable. We need a variable to represent the makespan (not a constant value) because we intend to trade off makespan values against resource capacity.

```
makespanVar = IloNumVar(env, 0, horizon, IloNumVar::Int);
```

- It creates the capacity of the resource as a variable as well for the same reason. We compute the maximal value of the `capacityVariable` as the sum, over all activities, of the required capacities. The *theoretical capacity* of the resource is set to this maximal value `capacity`.

```
capacityVar = IloNumVar(env, demandMax, capacity, IloNumVar::Int);
```

- It creates the resource as an instance of the predefined ILOG Scheduler class, `IloDiscreteResource`.

```
IloDiscreteResource resource(env, capacity);
```

- It creates a fake activity that requires `capacity-capacityVariable` units of the resource.

```
IloActivity rest(env, horizon);
rest.setStartTime(0);
IloNumVar qtyVar(env, 0, IloIntMax, IloNumVar::Int);
model.add(qtyVar == capacity - capacityVar);
model.add(rest.requires(resource, qtyVar));
char buffer[128];
sprintf(buffer, “Activity %d “, 0);
rest.setName(buffer);
```

- It uses the predefined class `IloActivity` to create an array of (real) activities and names those activities so that it will be easy to refer to them in a printed solution.

```
IloActivity* activities =
  new IloActivity[numberOfActivities];
for (i = 0; i < numberOfActivities; i++) {
  char name[128];
  sprintf(name, “Activity %d “, i+1);
  activities[i] = IloActivity(env, durations[i], name);
  model.add(activities[i].requires(resource, demands[i]));
  model.add(activities[i].getEndVariable() <= makespanVar);
}
```

- It posts the precedence constraints among the activities in the model.

```
IloInt precedenceIndex;
for (precedenceIndex = 0; ; precedenceIndex = precedenceIndex + 2) {
  IloInt predNumber = precedences[precedenceIndex] - 1;
  if (predNumber == -1)
    break;
  IloInt succNumber = precedences[precedenceIndex + 1] - 1;
  model.add(activities[succNumber].startsAfterEnd(activities[predNumber]));
}
```

- It deletes the array of activities.

```
delete [] activities;
```

Solving the Problem

As in other Concert Technology applications, we first create the environment for the problem and declare the numeric variables that interest us (`makespanVar` and `capacityVar`). Then we call our user-defined function `DefineProblem` to create a model specific to this problem.

```
IloEnv env;
IloNumVar makespanVar;
IloNumVar capacityVar;
IloModel model = DefineProblem(env,
                               NumberOfActivities,
                               Durations,
                               Demands,
                               Precedences,
                               makespanVar,
                               capacityVar);
```

Defining Two Objectives

The heart of our application is our implementation of an algorithm for Pareto-optimal trade-offs between two criteria: the makespan and resource capacity. To set up those trade-offs, we define two objectives in our model. One objective minimizes the makespan.

```
IloObjective minMakespanObj = IloMinimize(makespanVar);
IloGoal minMakespanGoal =
  IloSetTimesForward(env, makespanVar, IloSelFirstActMinEndMin);
```

The other objective minimizes the resource capacity.

```
IloObjective minCapacityObj = IloMinimize(capacityVar);
IloGoal minCapacityGoal =
  IloSetTimesForward(env, capacityVar, IloSelFirstActMinEndMin);
```

Both objectives use the predefined Concert Technology function `IloMinimize` to indicate the sense of the objective.

Each objective also uses a *goal* (an instance of `IloGoal`) to guide the search. Those goals are based on the predefined ILOG Scheduler function `IloSetTimesForward` that assigns a start time to activities in a model. It assigns start times on the basis of the predefined criterion `IloSelFirstActMinEndMin`, documented in the *ILOG Scheduler Reference*

Manual. Briefly, this criterion selects the first activity that has not been postponed, that has the minimal earliest start time (so it may possibly begin first), and that has the earliest end time (so it increases our makespan—the overall duration of the schedule—as little as possible).

Understanding the Algorithm for Pareto-Optimal Trade-offs

The following algorithm determines all the Pareto-optimal trade-offs between two optimization criteria, C_1 and C_2:

1. Generate a solution which minimizes the first criterion C_1. (If there is no solution at all, exit.) Let V_1 be the optimal value for C_1.
2. Constrain C_1 to be equal to V_1, and generate a solution which minimizes the second criterion, C_2. Let V_2 be the optimal value for C_2. The solution found will be a Pareto-optimal solution to the problem.
3. Remove the constraint stating that C_1 equals V_1. Replace it by a constraint stating that C_2 must be strictly less than V_2. Go to step 1.

It is easy to verify that the algorithm determines all Pareto-optimal trade-offs; that is, that for every Pareto-optimal solution S, the algorithm generates either S or a solution equivalent to S with respect to both C_1 and C_2. Here's how we verify that point: assume `(C1(S) C2(S))` is a Pareto-optimal trade-off. Then two cases can occur:

- First case: $(C_1(S)\ C_2(S))$ is the Pareto-optimal trade-off for which $C_2(S)$ is maximal. Then $C_1(S)$ is minimal and the solution S' generated at the first execution of step 2 satisfies $C_1(S') = C_1(S)$ and $C_2(S') = C_2(S)$.
- Second case: $(C_1(S)\ C_2(S))$ is not the Pareto-optimal trade-off for which $C_2(S)$ is maximal. In this case, let $(C_1(S')\ C_2(S'))$ be a Pareto-optimal trade-off which satisfies the following conditions:

  (a) $(C_1(S')\ C_2(S'))$ is found by the algorithm;

  (b) $C_2(S) < C_2(S')$; and

  (c) $C_2(S')$ is minimal among all the trade-offs that satisfy (a) and (b).
- At the iteration following the generation of S', the algorithm generates a Pareto-optimal solution S'' with $C_2(S'') < C_2(S')$. S'' satisfies (a), so it cannot satisfy (b). Hence, $C_2(S'') \leq C_2(S)$. Hence, $C_1(S) \leq C_1(S'')$ as both S and S'' are Pareto-optimal. But then necessarily $C_1(S) = C_1(S'')$; otherwise, $C_1(S'')$ would not be a possible value for S'' at step 2. Hence, $C_1(S) = C_1(S'')$ and $C_2(S) = C_2(S'')$ and the Pareto-optimal trade-off $(C_1(S)\ C_2(S))$ has been found.

Implementing the Pareto-Optimal Algorithm

As you surmised from the skeleton application in *A Preview*, we follow the same procedure for each Pareto criterion in turn. First, for the *capacity* criterion,

- we add a *capacity* constraint to the model;

```
model.add(minCapacityCt);
```

- we add the *makespan* objective to the model;

```
model.add(minMakespanObj);
```

We solve for the optimal solution with respect to the second criterion (that is, the shortest makespan).

```
if (solver.solve(minMakespanGoal))
  makespanValue = solver.getIntVar(makespanVar).getValue();
else
  break;
```

Then we remove the objective for *that* criterion and replace it in our model with the objective of the *other* criterion.

```
model.remove(minMakespanObj);
model.add(minCapacityObj);
```

Between these two criteria (shortest possible makespan and least resource capacity), we post a constraint to measure the trade-offs and add it to our model as well, like this:

```
IloConstraint isoMakespanCt = (makespanVar == makespanValue);
model.add(isoMakespanCt);
```

Once we have successfully solved both parts of the problem, we remove certain of those constraints and objectives, like this:

```
model.remove(isoMakespanCt);
model.remove(minCapacityCt);
model.remove(minCapacityObj);
minCapacityCt = (capacityVar < capacityValue);
```

Displaying a Solution

To display the solutions we find, we define the function `PrintSolution`.

```
void
PrintSolution(IloSolver solver)
{
  IlcScheduler scheduler(solver);
  for(IlcActivityIterator iterator(scheduler);
      iterator.ok();
      ++iterator)
    solver.out() << *iterator << endl;
}
```

Given an instance of `IloSolver`, this user-defined function exploits a predefined iterator (documented in the *ILOG Scheduler Reference Manual*) to scan the activities in the schedule. For each activity, it then uses the output facilities of `IloSolver` to display an identifier (such as `Activity 2`), along with its start time, duration, and end time, like this:

```
Activity 2  [3 -- 4 --> 7]
```

Ending an Application

As in all Concert Technology applications, we conclude this program by calling the member function `IloEnv::end` in order to clean up memory allocations for the environment and the model.

```
  env.end();
```

Complete Program

You can see the complete program here or on line in the standard distribution of ILOG Scheduler in the file `/examples/src/shippar.cpp`. To run this example, you need a license for ILOG Solver and Scheduler.

```
#include <ilsched/iloscheduler.h>

ILOSTLBEGIN

IloInt NumberOfActivities = 34;
IloInt Durations [] = {3, 4, 4, 6, 5, 2, 3, 4, 3, 2,
                       3, 2, 1, 5, 2, 3, 2, 2, 1, 1,
                       1, 2, 4, 5, 2, 1, 1, 2, 1, 3,
                       2, 1, 2, 2};
```

```
IloInt Demands [] = {4, 4, 3, 4, 5, 5, 4, 3, 4, 8,
                     4, 5, 4, 3, 3, 3, 6, 7, 4, 4,
                     4, 4, 7, 8, 8, 3, 3, 6, 8, 3,
                     3, 3, 3, 3 };
IloInt Precedences [] = {1, 2, 1, 4,
                         2, 3,
                         3, 5, 3, 7,
                         4, 5,
                         5, 6,
                         6, 8,
                         7, 8,
                         8, 9,
                         9, 10, 9, 14,
                         10, 11, 10, 12,
                         11, 13,
                         12, 13,
                         13, 15, 13, 16,
                         14, 15,
                         15, 18,
                         16, 17,
                         17, 18,
                         18, 19, 18, 20, 18, 21,
                         19, 23,
                         20, 23,
                         21, 22,
                         22, 23,
                         23, 24,
                         24, 25,
                         25, 26, 25, 30, 25, 31, 25, 32,
                         26, 27,
                         27, 28,
                         28, 29,
                         30, 28,
                         31, 28,
                         32, 33,
                         33, 34,
                         0, 0};

////////////////////////////////////////////////////////////////////////////
//
// PROBLEM DEFINITION
//
////////////////////////////////////////////////////////////////////////////
IloModel
DefineProblem(IloEnv env,
              IloInt numberOfActivities,
              IloInt* durations,
              IloInt* demands,
              IloInt* precedences,
              IloNumVar& makespanVar,
              IloNumVar& capacityVar)
{
  /* CREATE THE SCHEDULE. */
```

```
  IloInt horizon = 0;
  IloInt capacity = 0;
  IloInt i;
  IloNum demandMax = 0;
  for (i = 0; i < numberOfActivities; i++) {
    horizon += durations[i];
    capacity += demands[i];
    demandMax = IloMax(demandMax, demands[i]);
  }
  IloModel model(env);
  /* CREATE THE MAKESPAN VARIABLE. */
  makespanVar = IloNumVar(env, 0, horizon, IloNumVar::Int);
  /* CREATE THE CAPACITY VARIABLE. */
  capacityVar = IloNumVar(env, demandMax, capacity, IloNumVar::Int);
  /* CREATE THE RESOURCE. */
  IloDiscreteResource resource(env, capacity);
  /* CREATE THE FAKE “REST” ACTIVITY. */
  IloActivity rest(env, horizon);
  rest.setStartTime(0);
  IloNumVar qtyVar(env, 0, IloIntMax, IloNumVar::Int);
  model.add(qtyVar == capacity - capacityVar);
  model.add(rest.requires(resource, qtyVar));
  char buffer[128];
  sprintf(buffer, “Activity %d “, 0);
  rest.setName(buffer);

 /* CREATE THE ACTIVITIES. */
  IloActivity* activities =
    new IloActivity[numberOfActivities];
  for (i = 0; i < numberOfActivities; i++) {
    char name[128];
    sprintf(name, “Activity %d “, i+1);
    activities[i] = IloActivity(env, durations[i], name);
    model.add(activities[i].requires(resource, demands[i]));
    model.add(activities[i].endsBefore(makespanVar));
  }
  /* POST THE PRECEDENCE CONSTRAINTS. */
  IloInt precedenceIndex;
  for (precedenceIndex = 0; ; precedenceIndex = precedenceIndex + 2) {
    IloInt predNumber = precedences[precedenceIndex] - 1;
    if (predNumber == -1)
      break;
    IloInt succNumber = precedences[precedenceIndex + 1] - 1;
    model.add(activities[succNumber].startsAfterEnd(activities[predNumber]));
  }
  /* DELETE THE ARRAY OF ACTIVITIES. */
  delete [] activities;

  /* RETURN THE CREATED MODEL. */
  return model;
}
```

```
////////////////////////////////////////////////////////////////////////////
//
// PRINTING OF SOLUTIONS
//
////////////////////////////////////////////////////////////////////////////

void
PrintSolution(IloSolver solver)
{
  IlcScheduler scheduler(solver);
  for(IlcActivityIterator iterator(scheduler);
      iterator.ok();
      ++iterator)
    solver.out() << *iterator << endl;
}

////////////////////////////////////////////////////////////////////////////
//
// MAIN FUNCTION
//
////////////////////////////////////////////////////////////////////////////

int main()
{
  try {
    IloEnv env;
    IloNumVar makespanVar;
    IloNumVar capacityVar;
    IloModel model = DefineProblem(env,
                                   NumberOfActivities,
                                   Durations,
                                   Demands,
                                   Precedences,
                                   makespanVar,
                                   capacityVar);
    /* SOLVE THE PARETO SCHEDULING PROBLEM */
    IloSolver solver(model);

    IloObjective minMakespanObj = IloMinimize(env, makespanVar);
    IloGoal minMakespanGoal =
      IloSetTimesForward(env, makespanVar, IloSelFirstActMinEndMin);

    IloObjective minCapacityObj = IloMinimize(env, capacityVar);
    IloGoal minCapacityGoal =
    IloSetTimesForward(env, capacityVar, IloSelFirstActMinEndMin);

    IloInt makespanValue;
    IloInt capacityValue = IloIntMax;

    IloConstraint minCapacityCt = (capacityVar < capacityValue);

    while (IloTrue) {
      model.add(minCapacityCt);
```

```
      model.add(minMakespanObj);

      if (solver.solve(minMakespanGoal))
        makespanValue = solver.getIntVar(makespanVar).getValue();
      else
        break;

      model.remove(minMakespanObj);
      model.add(minCapacityObj);

      IloConstraint isoMakespanCt = (makespanVar == makespanValue);
      model.add(isoMakespanCt);

      if (solver.solve(minCapacityGoal)) {
        capacityValue = solver.getIntVar(capacityVar).getValue();
        solver.out() << "NEW PARETO OPTIMAL SOLUTION" << endl;
        solver.out() << "PRIMARY CRITERION = "   << makespanValue << endl;
        solver.out() << "SECONDARY CRITERION = " << capacityValue << endl;
        PrintSolution(solver);
      } else
        break;

      model.remove(isoMakespanCt);
      model.remove(minCapacityCt);
      model.remove(minCapacityObj);
      minCapacityCt = (capacityVar < capacityValue);
    }
    env.end();
  } catch (IloException& exc) {
    cout << exc << endl;
  }
  return 0;
}
```

Results

Here are the results of that program. Its execution provides the three Pareto-optimal trade-offs for the ship-loading problem:

- 66 units of time for maximal capacity 8;
- 59 units of time for maximal capacity 9;
- 58 units of time for maximal capacity 11.

These results illustrate how useful intermediate compromises can be. For example, if we think of a unit of capacity as a dockworker, and a unit of time as an hour worked, we can see from comparing these results that a project manager might be willing to add one unit of

capacity in order to save seven units of time, but probably would not want to add two additional units of capacity in order to save one more unit of time.

```
NEW PARETO OPTIMAL SOLUTION
PRIMARY CRITERION = 58
SECONDARY CRITERION = 11
Activity 34  [56 -- 2 --> 58]
Activity 33  [54 -- 2 --> 56]
Activity 32  [52 -- 1 --> 53]
Activity 31  [53 -- 2 --> 55]
Activity 30  [52 -- 3 --> 55]
Activity 29  [57 -- 1 --> 58]
Activity 28  [55 -- 2 --> 57]
Activity 27  [53 -- 1 --> 54]
Activity 26  [52 -- 1 --> 53]
Activity 25  [50 -- 2 --> 52]
Activity 24  [45 -- 5 --> 50]
Activity 23  [41 -- 4 --> 45]
Activity 22  [39 -- 2 --> 41]
Activity 21  [38 -- 1 --> 39]
Activity 20  [38 -- 1 --> 39]
Activity 19  [39 -- 1 --> 40]
Activity 18  [36 -- 2 --> 38]
Activity 17  [34 -- 2 --> 36]
Activity 16  [31 -- 3 --> 34]
Activity 15  [34 -- 2 --> 36]
Activity 14  [29 -- 5 --> 34]
Activity 13  [30 -- 1 --> 31]
Activity 12  [27 -- 2 --> 29]
Activity 11  [27 -- 3 --> 30]
Activity 10  [25 -- 2 --> 27]
Activity 9  [22 -- 3 --> 25]
Activity 8  [18 -- 4 --> 22]
Activity 7  [11 -- 3 --> 14]
Activity 6  [16 -- 2 --> 18]
Activity 5  [11 -- 5 --> 16]
Activity 4  [3 -- 6 --> 9]
Activity 3  [7 -- 4 --> 11]
Activity 2  [3 -- 4 --> 7]
Activity 1  [0 -- 3 --> 3]
Activity 0  [0 -- 87 --> 87]
NEW PARETO OPTIMAL SOLUTION
PRIMARY CRITERION = 59
SECONDARY CRITERION = 9
Activity 34  [56 -- 2 --> 58]
Activity 33  [54 -- 2 --> 56]
Activity 32  [52 -- 1 --> 53]
Activity 31  [52 -- 2 --> 54]
Activity 30  [53 -- 3 --> 56]
Activity 29  [58 -- 1 --> 59]
Activity 28  [56 -- 2 --> 58]
Activity 27  [53 -- 1 --> 54]
Activity 26  [52 -- 1 --> 53]
```

```
Activity 25  [50 -- 2 --> 52]
Activity 24  [45 -- 5 --> 50]
Activity 23  [41 -- 4 --> 45]
Activity 22  [39 -- 2 --> 41]
Activity 21  [38 -- 1 --> 39]
Activity 20  [38 -- 1 --> 39]
Activity 19  [39 -- 1 --> 40]
Activity 18  [36 -- 2 --> 38]
Activity 17  [34 -- 2 --> 36]
Activity 16  [31 -- 3 --> 34]
Activity 15  [34 -- 2 --> 36]
Activity 14  [29 -- 5 --> 34]
Activity 13  [30 -- 1 --> 31]
Activity 12  [27 -- 2 --> 29]
Activity 11  [27 -- 3 --> 30]
Activity 10  [25 -- 2 --> 27]
Activity 9  [22 -- 3 --> 25]
Activity 8  [18 -- 4 --> 22]
Activity 7  [11 -- 3 --> 14]
Activity 6  [16 -- 2 --> 18]
Activity 5  [11 -- 5 --> 16]
Activity 4  [3 -- 6 --> 9]
Activity 3  [7 -- 4 --> 11]
Activity 2  [3 -- 4 --> 7]
Activity 1  [0 -- 3 --> 3]
Activity 0  [0 -- 87 --> 87]
NEW PARETO OPTIMAL SOLUTION
PRIMARY CRITERION = 66
SECONDARY CRITERION = 8
Activity 34  [61 -- 2 --> 63]
Activity 33  [58 -- 2 --> 60]
Activity 32  [57 -- 1 --> 58]
Activity 31  [59 -- 2 --> 61]
Activity 30  [60 -- 3 --> 63]
Activity 29  [65 -- 1 --> 66]
Activity 28  [63 -- 2 --> 65]
Activity 27  [58 -- 1 --> 59]
Activity 26  [57 -- 1 --> 58]
Activity 25  [55 -- 2 --> 57]
Activity 24  [50 -- 5 --> 55]
Activity 23  [46 -- 4 --> 50]
Activity 22  [44 -- 2 --> 46]
Activity 21  [43 -- 1 --> 44]
Activity 20  [43 -- 1 --> 44]
Activity 19  [44 -- 1 --> 45]
Activity 18  [41 -- 2 --> 43]
Activity 17  [39 -- 2 --> 41]
Activity 16  [36 -- 3 --> 39]
Activity 15  [36 -- 2 --> 38]
Activity 14  [30 -- 5 --> 35]
Activity 13  [35 -- 1 --> 36]
Activity 12  [30 -- 2 --> 32]
```

```
Activity 11  [32 -- 3 --> 35]
Activity 10  [28 -- 2 --> 30]
Activity 9  [25 -- 3 --> 28]
Activity 8  [21 -- 4 --> 25]
Activity 7  [11 -- 3 --> 14]
Activity 6  [19 -- 2 --> 21]
Activity 5  [14 -- 5 --> 19]
Activity 4  [3 -- 6 --> 9]
Activity 3  [7 -- 4 --> 11]
Activity 2  [3 -- 4 --> 7]
Activity 1  [0 -- 3 --> 3]
Activity 0  [0 -- 87 --> 87]
```

Index